# Probabilistic program analysis

Kirkeby, Maja Hanne

*Publication date:*
2018

*Document Version*
Publisher's PDF, also known as Version of record

*Citation for published version (APA):*
Kirkeby, M. H. (2018). *Probabilistic program analysis*. Roskilde Universitet. Roskilde Universitet. Computer Science. Computer Science Research Report Vol. 151

# Probabilistic Program Analysis

Maja Hanne Kirkeby




JUNE 2018 ROSKILDE UNIVERSITY COMPUTER SCIENCE RESEARCH REPORT #151


Computer Science
Roskilde University
P. O. Box 260
DK–4000 Roskilde
Denmark

http://www.ruc.dk/



ISSN 0109–9779

Research reports are available electronically from:

http://ojs.ruc.dk/index.php/csrr/issue/archive

Maja Hanne Kirkeby

# Probabilistic Program Analysis

June 2018

*Emilie's fish (2017)*

Supervisor: Mads Rosendahl

This dissertation is submitted to the
Department of People and Technology,
in partial fulfillment of the requirements for the degree of
**Doctor of Philosophy** in the subject of **Computer Science** .

Roskilde University,
Denmark

*To my loving husband Anders and daughter Emilie.*

# Abstract

This thesis addresses the challenge of inferring information about the probabilities of a program's output or its resource usage. Given the probabilities of the program's input, a deterministic program, and a set of output/resource usages, we want to analyze the probability of this set. In general, it is not always possible to compute precise probabilities, and our strategy is to apply different abstractions and compute a pair of upper and lower bounds of the probabilities.

We (i) present a transformation-based probability analysis for discrete probabilities yielding upper probability bounds of discrete output events, (ii) show how to apply the transformation-based probability analysis to resource instrumented programs and obtain upper probability bounds of discrete resource events, and (iii) present two approaches that reuse established non-probabilistic (forward or backwards) analyses and obtain upper and lower probability bounds for the output events. Furthermore, we (iv) present approaches to analyse non-deterministic and probabilistic programs; that is, we identify properties that if established ensure that the programs may be considered as functional, deterministic programs and are thereby amenable to apply the presented approaches of (iii).

# Abstract (Danish)

Denne afhandling analyserer sandsynligheder for forskellige hændelser relateret til et programs udførsel. En hændelse kan være bestemte output eller ressourceforbrug. Givet et deterministisk program, sandsynlighederne for programmets input, samt en gruppe af hændelser, det vil sige mængder af output eller ressourceforbrug, vil vi analysere sandsynlighederne for disse hændelser. Det er ikke altid muligt at beregne præcise sandsynligheder men ved at bruge forskellige abstraktioner kan vi beregne øvre og nedre grænser for hændelsernes sandsynligheder.

Vi præsenterer først en transformationsbaseret sandsynlighedsanalyse for diskrete input-sandsynligheder, der giver øvre sandsynlighedsgrænser for diskrete outputhændelser. Derefter viser vi hvordan man kan anvende denne transformationsbaserede sandsynlighedsanalyse på ressource-instrumenterede programmer og derved opnå øvre sandsynlighedsgrænser for diskrete ressourcehændelser. Herefter udvider vi fokus til det mere generelle tilfælde hvor programmernes input-sandsynlighederne ikke er begrænsede til det diskrete tilfælde. Vi præsenterer to metoder, der genbruger etablerede ikke-probabilistiske (forlæns eller baglæns orienterede) analyser til at beregne øvre og nedre sandsynlighedsgrænser for output-hændelserne. I den sidste del præsenterer vi forskellige tilgange til at analysere ikke-deterministiske og probabilistiske programmer. Målet her er således at identificere egenskaber, der, hvis de etableres, sikrer, at programmerne kan betragtes som funktionelle, deterministiske programmer og at man derved f.eks. kan anvende de to metoder, der genbruger eksisterende analyser.

# Acknowledgments

I would like to thank my supervisor, Mads Rosendahl, for the guidance and support during the project. I am grateful for the always intriguing scientific discussions and sharing of ideas, as well as your confidence in my progress as a researcher.

In addition, I would like to thank Henning Christiansen for all the scientific discussions and the always open door. In particular, of course, I would like to thank you for the exciting collaboration on the three articles concerning confluence (modulo equivalence) in both probabilistic systems and constraint handling rules.

It has been a privilege to perform research in the context of the Whole-Systems Energy-Transparency collaboration (ENTRA) for three years. It was a well-organized collaboration and both fun and developing to be a part of. I appreciate you all, and I would like to express my gratitude towards John Gallagher, Pedro Lopez-Garcia, Kerstin Eder, and Henk Muller for taking your time to provide both support and constructive feedback during the whole period. I would like to especially thank the local ENTRA group, Bishoksan Kafle, Nina Bohr, Xueliang Li, Morten Rhiger, Mads Rosendahl and John Gallagher, for providing encouraging scientific environment that facilitated maturation of ideas. I would also like to thank the ENTRA group in Madrid, Pedro-Lopez Garcia and Alejandro Serrano, for inspiring discussions on probabilities and Maximiliano Klemen, Remy Haermelle, and Umer Liqat for the discussions and excellent help on the Ciao(PP) system.

I acknowledge the importance of my colleagues in computer science and informatics for providing an inspiring and intellectual environment. I would like to thank Bishoksan Kafle for your encouragement and all the shared lunches, Maria Ie Manikas and Benedicte Florin for the countless number of productive work sessions. I would like to provide special acknowledgment to Torben Æ. Mogensen and Holger B. Axelsen for encouraging my research.

Furthermore, I would like to express my gratitude to my family and friends for always being there, to my parents and parents-in-law for your great help and support, to my inspiring and helpful brothers Asger and Rune and their families, and especially to my beloved husband Anders and my daughter Emilie for your care, your love and your support throughout this period.

Finally, I would like to express my gratitude to the assessment committee for your time and your invaluable comments and recommendations.

# Contents

# 1

# Introduction

In this thesis, we will develop static analyses that infer information about the probabilities of a program's output or its resource usage. We want to answer questions such as *"what is the probability that a given program yields a result between 0 and 10?"* and *"what is the probability that a given program uses more than 10 resource units,* e.g.*, time steps, before it stops?"*.

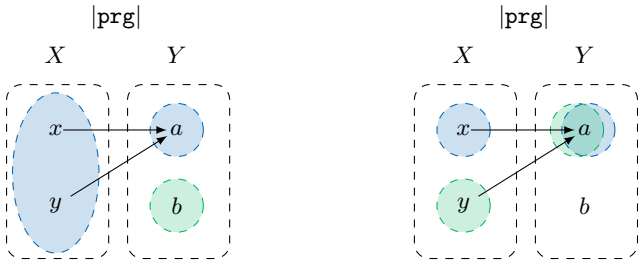
In the following, we will introduce the necessary concepts such as program semantics, properties, and probabilities. Afterwards, we will formulate the challenge of the thesis and describe the approaches that we take; the descriptions include pointers to their respective chapters. Then, we outline the research fields that form the basis of this thesis, and finally, we provide a list of our key contributions together with references to the related articles in which they are published.

## 1.1 Basic concepts of thesis

In this thesis, we consider the semantics of a program $\texttt{prg}$, namely $|\texttt{prg}|$, to be a relation between input $X$ and output $Y$, *i.e.*, a set of input-output pairs $|\texttt{prg}| \subseteq X \times Y$. For the special case where the program is instrumented with a resource model, the program semantics is a relation between input and output together with resource usage, a value from $R$, *i.e.*, a set of pairs $|\texttt{prg}| \subseteq X \times (Y \times R)$. For a deterministic program, the input-output relation is *functional*, *i.e.*, each input is related to at most one output. Programs that terminate for all inputs $x \in X$ define *total* relations, *i.e.*, each input relates to at least one output. Depending on the analysed language, $Y$ could contain special elements for program results that are not per se outputs, for instance, error or nontermination.

An *input property* is a subset of the input $X$, an *output property* is a subset of the output $Y$, and a *resource property* is a subset of the resource usage $R$. The input properties and output/resource properties are related through the program relation; each output/resource property $A$ has a *pre-image* $pre_{|\texttt{prg}|}(A)$, *i.e.*, the set of inputs that relate to an output occurring in $A$, $pre_{|\texttt{prg}|}(A) = \{x \mid \exists y \in A\colon (x, y) \in |\texttt{prg}|\}$. Moreover, every input property $A$ has an *image* $img_{|\texttt{prg}|}(A)$, *i.e.*, the outputs that are

related to some element in $A$, $img_{|\texttt{prg}|}(A) = \{y \mid \exists x \in A\colon (x, y) \in |\texttt{prg}|\}$. See Figure 1.1 for examples of pre-images and images.




(a) $pre_{|\texttt{prg}|}$ (b) $img_{|\texttt{prg}|}$

Fig. 1.1: The diagrams (arrows) depict the same total function $|\texttt{prg}|$ from the input set $X = \{x, y\}$ and to the output set $Y = \{a, b\}$. A color indicates the pre-image (a) and respectively the image (b) of the singleton event with the same color. For instance, in (a), the blue output event $\{a\}$ has the blue input event $\{x, y\}$ as the pre-image $pre_{|\texttt{prg}|}(\{a\}) = \{x, y\}$. Note that the pre-image of the green output event $\{b\}$ is the empty input event.

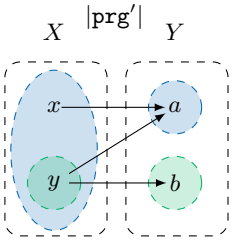
It may be the case that some inputs are more likely to be fed to the program; this can be described using probabilities. For the purpose of this thesis, an event is a property, namely, a set of inputs or outputs. Informally, a *probability measure* describes "the chance that a given event will occur" [90]; a probability measure over a set is a mapping from subsets of the set, the events, to their probability. In case there is a countable set (finite or infinite) with probability 1, a probability distribution, *i.e.*, a mapping from each individual element to its probability, suffices.

The probability of an output (or resource) property depends on (i) the pre-image, specifying which input leads to the output (or resource) property, and (ii) the probability that those inputs are fed to the program. To describe this, we define a function $P_{\texttt{prg}} = \lambda\mu.\lambda A.\mu(pre_{|\texttt{prg}|}(A))$, where $\mu$ is a probability measure over input events and $A$ is an output event; if $|\texttt{prg}| \subseteq X \times Y$ is a total function, *i.e.*, the program prg is deterministic, $pre_{|\texttt{prg}|}$ is the programs pre-image function, and $\mu_X$ is a probability measure over $X$, then $P_{|\texttt{prg}|}(\mu_X)$ can be shown to define a probability measure over output properties. However, more generally, when $|\texttt{prg}|$ is a total relation, for instance, when the program prg is nondeterministic, then $P_{\texttt{prg}}(\mu_X)$ is in general non-additive[1] and, thus, is not a probability measure, as demonstrated in Example 1.1. Instead, $P_{|\texttt{prg}|}(\mu_X)(A)$ gives an upper bound of the probability that the output event $A$ occurs.

*Example 1.1.* Let $\mu_X$ be a discrete probability measure defined by $\mu_X(\{x\}) = 70\%$ and $\mu_X(\{y\}) = 30\%$. Let $\texttt{prg}'$ be a nondeterministic program as defined in Fig-

[1] A function $f$ from a power set to the reals is *additive* if and only if, for all disjoint sets $A$ and $B$, $f(A \uplus B) = f(A) + f(B)$; otherwise, $f$ is *non-additive*.

ure 1.2a, where $|\texttt{prg}'|$ is a total relation with the pre-image function $pre_{|\texttt{prg}'|}$. The upper probability bound function of $\texttt{prg}'$, namely, $P_{\texttt{prg}'}(\mu_X)$, yields the bounds given in Figure 1.2b; here, the fact that $P_{\texttt{prg}'}(\mu_X)(\{a\}) + P_{\texttt{prg}'}(\mu_X)(\{b\}) \neq P_{\texttt{prg}'}(\mu_X)(\{a,b\})$ shows that $P_{\texttt{prg}'}(\mu_X)$ is non-additive and, thus, is not a probability measure.




(a)

| | $P_{\texttt{prg}'}(\mu_X)$ |
|---|---|
| $\emptyset$ | 0% |
| $\{a\}$ | 100% |
| $\{b\}$ | 30% |
| $\{a,b\}$ | 100% |

(b)

Fig. 1.2: (a): The relation $|\texttt{prg}'|$ is a total relation. (b): The discrete probability measure $\mu_X$ is defined by $\mu_x(\{x\}) = 70\%$ and $\mu_x(\{y\}) = 30\%$, and $P_{\texttt{prg}'}(\mu_x)$ defines upper probability bounds of the output events of $|\texttt{prg}|$.

The function $P_{\texttt{prg}}$ is a special case of a more general formulation $P'_{|\texttt{prg}|}$ of the upper probability bounds; the upper probability bound of an output event depends on (i) the inputs that *may* lead to the output and (ii) the probability that those inputs are fed to the program. We let $pre^{\sharp}_{|\texttt{prg}|}$ be a function that relates to the pre-image $pre_{|\texttt{prg}|}$ such that $pre_{|\texttt{prg}|}(A) \subseteq pre^{\sharp}_{|\texttt{prg}|}(A)$ and use that to define the more general $P'_{|\texttt{prg}|} = \lambda\mu.\lambda A.\mu(pre^{\sharp}_{|\texttt{prg}|}(A))$. There exists a $P'_{|\texttt{prg}|}$ for each $pre^{\sharp}_{|\texttt{prg}|}$. Such a $P^{\sharp}_{|\texttt{prg}|}(A) = P'_{|\texttt{prg}|}(\mu_X)(A)$ can be shown to be an upper bound of the probability that the output event $A$ occurs. Each $P^{\sharp}_{|\texttt{prg}|}$ determines not only the upper probability bounds of the output properties but also, as we will show in Theorem 5.10, their lower probability bounds; it defines a function $P^{\flat}_{|\texttt{prg}|}$, *i.e.*, $P^{\flat}_{|\texttt{prg}|}(A) = 1 - P^{\sharp}_{|\texttt{prg}|}(A^{\complement})$ from output events to their lower probability bounds. When $|\texttt{prg}|$ is total, $P^{\flat}_{|\texttt{prg}|}(A)$ can be shown to be the probability of the dual[2] (approximated) pre-image $pre^{\sharp}_{|\texttt{prg}|}(A)$, namely, $\widetilde{pre}^{\complement\sharp}_{|\texttt{prg}|}(A)$. For a functional and total $|\texttt{prg}|$ whereby $pre^{\sharp}_{|\texttt{prg}|} = pre_{|\texttt{prg}|}$, $P^{\flat}_{|\texttt{prg}|} = P^{\sharp}_{|\texttt{prg}|}$ is a probability measure.

A formal introduction to probability measures/distributions and inferences over functions will be given in Chapter 2.

[2] Two functions $f, \widetilde{f} \colon \wp(X) \to \wp(Y)$ are *dual* if and only if $\widetilde{f}(A) = f(A^{\complement})^{\complement}$.

## 1.2 Background and contributing fields

Related work is commented on and discussed when relevant throughout the chapters; see Chapters 3.8, 4.7,6.8, 7.6, and 8.4. Here, we summarize the four fields of research that form the background to the results presented in the thesis.

- One of the pillars is static analysis, which is analysis performed without executing the programs. A framework to describe such analysis is abstract interpretation [28, 29], which was introduced by Cousot and Cousot in 1997; corresponding subfields are cost analysis [6, 87] and complexity analysis [116, 141].
- Another foundation on which this thesis builds is probability theory, *e.g.* [72, 119], providing us with a "language" with which we can describe how often events occur. Probability theory can be dated back to the 16th century, but the modern definition using measurable spaces was formulated by Kolmogorov[3] in 1933 [81]. This theory relies on set theory, *e.g.* [73] In contrast to precise probabilities, the field of imprecise probabilities, *e.g.* [10, 49, 140, 147], generalizes probability theory so that we may approximate a probability measure by, for instance, sets of probability measures.
- The following fields build on both of the above fields; probabilistic semantics [42, 82], *i.e.*, semantics of programs that may contain random generators, and probabilistic analysis [4, 33, 41, 95, 120], which can be used to analyse such programs.
- The field of term rewriting systems, *e.g.* [5, 11, 34, 70, 105, 137], forms the basis for both the transformation-based nondeterministic language constraint handling rules [1, 2, 54, 55] and probabilistic abstract reduction systems [20], which of course also builds on probability theory.

The relations of the fields listed above to the individual chapters are summarized in Figure 1.3.

## 1.3 Challenge and Approaches

Thesis challenge:

> Given a deterministic program, a probability measure over the program inputs and an output/resource property, we want to analyse the probability of the output/resource property. In addition, we want to examine practical approaches to create such probabilistic analyses.

We focus on deterministic programs, but even for this class of programs, the probabilities of output properties are uncomputable in general, *i.e.* it is not always possible to compute a single probability measure for the desired output properties. Our strategy for attacking this problem is to introduce various abstractions and compute an upper bound $u$ and lower bound $l$ for each output or resource property $A \subseteq Y$, respectively $A \subseteq R$. In practice, we compute a function from events to a pair of upper

[3] Kolmogorov published in German under the name Kolmogoroff.

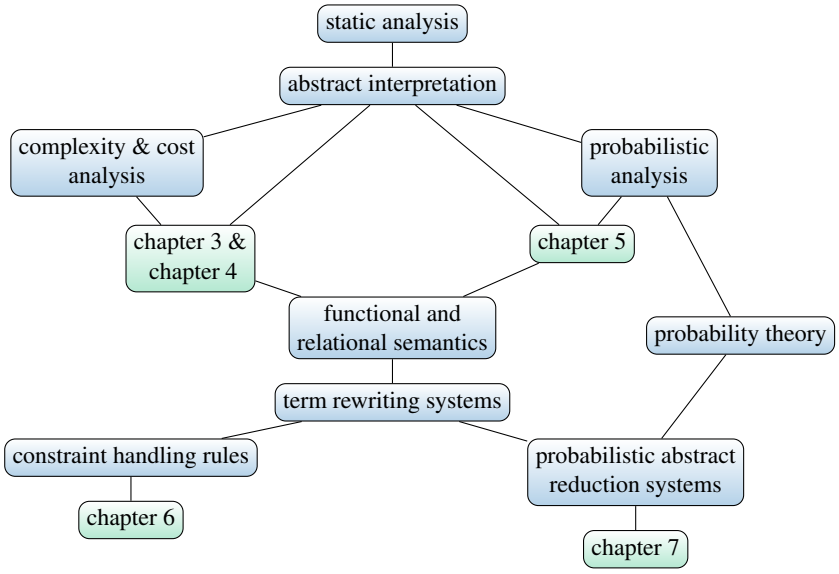



Fig. 1.3: Overview of the research fields that this thesis builds on.

and lower probability bounds $f(A) = (l, u)$ and let it define a set of probability measures, namely, $\{\mu \mid \mu \text{ is a prob. measure} \wedge \forall A\colon f(A) = (l,u) \Rightarrow l \leq \mu(A) \leq u\}$. We can always derive upper or lower bounds alone and simply consider the lower or upper bound, respectively, to be 0 or 1, respectively.

### 1.3.1 Thesis Outline

*Discrete output probability distributions.* In Chapter 3, we address discrete input and output measures that are perhaps parametrized, *e.g.* a uniform distribution between 0 and a natural number $n$. For a given output event $A$, we express $P_{\mathtt{prg}}(\mu)(A)$ as the sum of probabilities of all inputs that the program relates to the output, and we approximate it by manipulating and transforming this sum-expression or parts of it. The analysis stops when it reaches a closed-form expression, which in our case means an expression without summations, products or recursive expressions. The analysis draws upon recurrence equation methods known from resource analysis [109] and uses the computer algebra system Mathematica [146] to handle summations and products symbolically. The analysis yields upper probability bounds.

*Discrete resource probability distributions.* In Chapter 4, we apply these ideas to obtain a discrete probabilistic resource analysis that derives upper probability bounds for the resource usage of deterministic and terminating programs. First, we instrument the programs with a discrete and deterministic resource model and slice the program to output only resource usages. Then, we apply the previously developed probabilistic output analysis to obtain upper probability bounds for the resource usage. The model used in the examples describe the step counters.

*Output probability measures.* In Chapter 5, we take a step back and return to the general case wherein the input is described by a probability measure. We consider using established methods that provide over-approximations of the program's pre-image or image relation and present simple techniques that derive both upper and lower probability bounds for the output properties. Current analyses often provide a monotone function that we can use to define either a pre-image over-approximating function, *i.e.*, backwards analysis, or an image over-approximating function, *i.e.*, a forward analysis.

We show that an over-approximation of the pre-image $pre^\sharp(A)$ of an output event $A$ may be used to define a pair of upper and lower bounds in the same manner as $P^\sharp_{|\texttt{prg}|}$ and $P^\flat_{|\texttt{prg}|}$. The input set $pre^\sharp(A)$ of an output event $A$ may not be an input event, *i.e.*, $\mu_X$ may be undefined for this set, and we cannot use it to define $P'_{|\texttt{prg}|}$. Therefore, we introduce an abstraction $\uparrow$ that maps the input sets to input events (for which $\mu_X$ is defined) such that $pre^\sharp(A) \subseteq \ \uparrow pre^\sharp(A)$, and we let $\uparrow \circ\ pre^\sharp$ define $P'_{|\texttt{prg}|} = \lambda\mu.\lambda A.\mu((\uparrow \circ\ pre^\sharp)(A))$. This method requires a re-computation for each input probability measure and for each output property in which we are interested.

When the analysis instead provides a function $img^\sharp$ that over-approximates the images, *i.e.*, $img(A) \subseteq img^\sharp(A)$, we can indirectly determine a computable pre-image over-approximation $pre^\sharp$ and apply the "backwards" method. We recall that the image over singletons defines the program relation, which in return defines the pre-image, *i.e.*, $pre(A) \triangleq \{x \in X \mid img(\{x\}) \cap A \neq \emptyset\}$; a function that over-approximates the images $img^\sharp$ defines an over-approximation of the pre-image $pre^\sharp$ in the same way, *i.e.*, $pre^\sharp(A) \triangleq \{x \in X \mid img^\sharp(\{x\}) \cap A \neq \emptyset\}$. However, the input set may be infinite or insufficiently large such that a computation of such a pre-image over-approximation becomes intractable. In these cases, we instead suggest using a partition[4] $T$ of the inputs $X$ and defining a (perhaps less accurate) over-approximating pre-image of an output $A$ as the set of partition elements whose image overlaps with $A$, *i.e.*, $pre^\sharp[T](A) \triangleq \bigcup\{t \in T \mid img^\sharp(t) \cap A \neq \emptyset\}$.

We exemplify how to apply the method for the forward analysis and illustrate how the choice of the partition impacts the precision and computation cost; for this, we use two well-known analyses: a sign analysis and an interval analysis.

#### 1.3.1.1 Other Directions

We have previously focused on deterministic programs, and in this section, we broaden the perspective to programs that are not necessarily deterministic yet still have a functional input-output relation. When programs have functional input-output relations, we can re-use the framework discussed above, and we may, for instance, apply the methods discussed in Chapter 5 to infer probability bounds for their output properties.

*Proving Confluence.* Hence, we focus on proving program properties that ensure that the program semantics (or an abstraction thereof) is a total function. We draw

[4] A partition over a set $X$ is a set of pairwise disjoint subsets of $X$ that covers $X$.

on proof methods from Term Rewriting Systems and, more generally, Abstract Reduction Systems. Specifically, we make use of variations of the important property *confluence*, which is the property whereby, for each state that can reduce into different states, there exist another state to which all these reductions may lead. For terminating programs, confluence implies that all alternative program runs for a single input lead to the exact same output, ensuring a total and functional program relation. Furthermore, for terminating Term Rewriting Systems, this has been shown to be decidable [70], *i.e*. there exists a terminating algorithm that can correctly decide whether a program is confluent. In the following, we will build on these results.

*Confluence modulo equivalence in terminating and nondeterministic programs.* In Chapter 6, we address terminating nondeterministic programs in the form of programs written in Constraint Handling Rules (CHR), a nondeterministic committed-choice language [54, 55]. By definition, any program has a total and functional program relation if, for any single input, all alternative program runs lead to an output and always to the same output, *e.g*. the program is terminating and confluent. Confluence has also been shown to be decidable for terminating CHR programs [1]. Here, we will focus on a more general type of confluence, where "the same output" may be different but equally acceptable, *e.g*. a set may be represented by any list containing the set elements. Such nondeterministic programs, which are intended to compute any of the equally acceptable results, are typically not confluent but may yet be confluent modulo an equivalence (tailored for the given program). For terminating programs, a program is *confluent modulo equivalence* if all alternative program runs for the same input only lead to outputs that are equivalent.

In the context of probabilistic analysis, the outputs are abstracted into non-overlapping sets of equivalent output, and confluence modulo equivalence implies a functional relation from input to these sets of outputs. In case the output properties of interest can be constructed by a union of these sets/events, there exists a probability measure over these output properties.

*Almost-sure convergence in nonterminating probabilistic programs.* In Chapter 7, we focus on a different set of programs, namely, nonterminating probabilistic programs, here in the form of Probabilistic Abstract Reduction Systems. A *probabilistic program* can be seen as a nondeterministic program whereby the nondeterministic behavior is always constrained by a probability. The semantics of a probabilistic program is, in general, a relation, where for each input there exists a (sub-)probability measure[5] over its related output events ; we call such a measure the *related output measure*. When an input yields a nonterminating computation, its related output measure is a sub-probability measure, and when it yields only terminating computations, its related output measure is a probability measure. In short, the probability of an output event depends on not only (i) the inputs that may lead to the output event and (ii) the probability measure over input but also (iii) each inputs' related output measure. For instance, for the discrete case, the probability of an output event $A$ is a sum over all inputs in the pre-image of $A$, namely, $pre_{|\mathtt{prg}|}(A)$, where the input probability of

[5] A sub-probability measure is a measure with a positive total weight $\leq 1$, *e.g*. a probability measure is a sub-probability measure.

each input $\mu_X(\{x\})$ is multiplied by its chance of reaching $A$ as dictated by each related output measure $\mu_x(A)$, *i.e.* $\sum_{x \in pre_{|\mathtt{prg}|}(A)} \mu_X(\{x\}) \cdot \mu_x(A)$.

We look for program properties that ensure that the program semantics is total and functional. Such a property is *almost-surely convergence*, that is, a program is *almost-surely convergent* if for each input it reaches a unique output element with probability 1. We prove that a program is almost-surely convergent if the program is both confluent and *almost-surely terminating*, *i.e.* it terminates with probability 1. If a program is almost-surely convergent, then each input relates to one and only one output with probability 1. Consequently, (i) the input-output relation must be total since each input relates to an output, (ii) it must be functional since each input relates to only one output, and (iii) all the related output measures are trivial probability measures where the output is reached with probability 1. Therefore, we may ignore the related output measures and reduce the semantics of probabilistic almost-surely convergent programs to total functions. Together, this ensures that $P_{\mathtt{prg}}$ yields a probability measure over output events.

## 1.4 Contributions

The key contributions of this dissertation are as follows:

- We design an automatic discrete probabilistic output analysis for simple functional programs with a fixed and perhaps parametrized input probability distribution. The analysis results in a function from output elements to their upper probability bound.

  Published in
  [118] M. Rosendahl, M. H. Kirkeby. Probabilistic Output Analysis by Program Manipulation. *Electronic Proceedings in Theoretical Computer Science*, 194(318337):110–124, 2015

- Based on this probabilistic output analysis, we present a discrete probabilistic resource analysis for C-like programs. Assuming a discrete resource model and an input probability distribution, we first instrument the C-like program with the resource model, slice the instrumented program with respect to the resource usage and translate the sliced program into the functional language. Then, we apply the discrete probability output analysis to the now resource-computing functional program and the input probability distribution. The analysis returns a function from the resource properties of the given program to their upper probability bounds.

  Published in
  [78] M. H. Kirkeby, M. Rosendahl. Probabilistic resource analysis by program transformation. M. van Eekelen, U. Dal Lago, redaktorzy, *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, strony 60–80, Cham, 2016. Springer International Publishing

- We present two techniques, which given an existing input-output analysis and a probabilistic input measure provide upper and lower probability bounds for the output events. One technique covers backwards analysis, and one technique covers forward analysis. We prove the correctness of both and demonstrate the forward analysis via three case studies, one being interval analysis. When comparing our results with cutting-edge probabilistic analyses that can analyse both probabilistic programs, *e.g.* programs with random generators, and deterministic programs, we see that, for deterministic programs, this new technique yields results that are as good or better.
- We present approaches to the analysis of non-deterministic and probabilistic programs. In particular, we identify the properties of confluence modulo equivalence and almost-sure convergence, which if established allow such programs to be treated as functional, deterministic programs, hence being amenable to the analyses outlined above.

Published in the following articles.

[77] M. H. Kirkeby, H. Christiansen. Confluence and convergence in probabilistically terminating reduction systems. *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017*, wolumen abs/1709.05123, 2017. (accepted for publication)

[25] H. Christiansen, M. H. Kirkeby. Confluence Modulo Equivalence in Constraint Handling Rules. wolumen 8981, strony 41–58. Springer International Publishing Switzerland, 2015

# 2

# Probabilities and other prerequisites

In this chapter, we give a formal introduction to probability distributions and measures to describe the probability of input events. Then, we describe how to infer output probability distributions and measures via program relations when these describe a function. Both of these sections introduce standard probability theory, *e.g.* [59]. Finally, we introduce abstract interpretation [28, 29], *i.e.*, a framework for constructing program analyses.

For the definitions of relations and functions and their properties, we refer to Appendix A. It is important to emphasize that functions are assumed total. We will start with some notation:

$\wp(X)$ denotes the power set of the set $X$, *i.e.* the set of all subsets of $X$. The set $X$ is the *base* of the power set.
$A_1, A_2, \ldots A_k$ denotes a finite series of $k$ elements.
$A_1, A_2, \ldots$ denotes a countable infinite series.
$A^{\complement}$ denotes the complement of $A$ with respect to some set, which will be clearly indicated in the context.

## 2.1 Probabilistic Measures and Distributions

For programs over integers (or any other countable set), we can use a special function, namely, a probability distribution, to describe the probability that each integer will be fed to the program.

**Definition 2.1.** *A* probability distribution *is defined by a function* $P\colon X \to [0,1]$ *over a countable set* $X$ *for which* $\sum_{x\in X} P(x) = 1$.

In the following, we describe $\sigma$-algebras and measurable spaces, which are the event spaces that measures and, thus, probability measures are defined over. Probability distributions and probability measures are related in that a probability distribution uniquely describes a family of probability measures with similar behavior but differ in the set of events for which they are defined (see Proposition 2.20).

**Definition 2.2.** *A* $\sigma$-algebra $\mathcal{X}$ *is a non-empty subset of* $\wp(X)$ *that is closed under countable unions and complements,* i.e. *if* $A_1, A_2, \ldots \in \mathcal{X}$*, then* $\bigcup_{n=0}^{\infty} A_n \in \mathcal{X}$*, and if* $A \in \mathcal{X}$*, then* $A^{\complement} \in \mathcal{X}$.

*Example 2.3.* The sets $\{\emptyset, \{x, y\}\}$ and $\{\emptyset, \{x, y\}, \{x\}, \{y\}\}$ are $\sigma$-algebras, whereas neither $\{\emptyset, \{x, y\}, \{x\}\}$ nor $\{\{x\}, \{y\}\}$ are.

Let $X$ be a non-empty set; then, $\{\emptyset, X\}$ is a $\sigma$-algebra. This type of $\sigma$-algebra is often referred to as the *trivial* $\sigma$*-algebra* of $X$.

**Proposition 2.4.** *If* $\mathcal{X}$ *is a* $\sigma$*-algebra over* $X$*, then* $X \in \mathcal{X}$ *and* $\emptyset \in \mathcal{X}$

*Proof.* A $\sigma$-algebra is non-empty, thus, there is an $A \subseteq X$, such that $A \in \mathcal{X}$. By the definition of $\sigma$-algebras, $A \in \mathcal{X} \Rightarrow A^{\complement} \in \mathcal{X} \Rightarrow A^{\complement} \cup A \in \mathcal{X} \Rightarrow X \in \mathcal{X}$, and furthermore, $X \in \mathcal{X} \Rightarrow \emptyset \in \mathcal{X}$.

**Proposition 2.5.** *A* $\sigma$*-algebra* $\mathcal{X}$ *is closed under countable intersections,* i.e. *if* $A_1, A_2, \ldots \in \mathcal{X}$*, then* $\bigcap_{n=0}^{\infty} A_n \in \mathcal{X}$.

*Proof.* By the definition of the $\sigma$-algebra, $A_n \in \mathcal{X} \Rightarrow {A_n}^{\complement} \in \mathcal{X} \Rightarrow \bigcup_{n=0}^{\infty} {A_n}^{\complement} \in \mathcal{X} \Rightarrow \left(\bigcup_{n=0}^{\infty} {A_n}^{\complement}\right)^{\complement} \in \mathcal{X}$ and by de Morgans law $\left(\bigcup_{n=0}^{\infty} {A_n}^{\complement}\right)^{\complement} \in \mathcal{X} \Leftrightarrow \bigcap_{n=0}^{\infty} \left({A_n}^{\complement}\right)^{\complement} \in \mathcal{X} \Leftrightarrow \bigcap_{n=0}^{\infty} A_n \in \mathcal{X}$.

**Lemma 2.6.** *Let* $X$ *be a set, and let* $\mathbf{C}$ *be a collection of* $\sigma$*-algebras over* $X$*; then,* $\bigcap \mathbf{C}$ *is a* $\sigma$*-algebra over* $X$.

*Proof.* The proof is nearly trivial using the definitions of the $\sigma$-algebra and intersection, *i.e.*, $A \in \bigcap \mathbf{C} \Leftrightarrow \forall \mathcal{X} \in \mathbf{C} \colon A \in \mathcal{X}$; only the first axiom deviates. *non-empty:* By Proposition 2.4, $\forall \mathcal{X} \in \mathbf{C} \colon X \in \mathcal{X}$; thus, $\forall \mathcal{X} \in \mathbf{C} \colon X \in \mathcal{X} \Rightarrow X \in \bigcap \mathbf{C}$. *closed under complements*: $A \in \bigcap \mathbf{C} \Rightarrow \forall \mathcal{X} \in \mathbf{C} \colon A \in \mathcal{X} \Rightarrow \forall \mathcal{X} \in \mathbf{C} \colon A^{\complement} \in \mathcal{X} \Rightarrow A^{\complement} \in \cap \mathbf{C}$ *closed under countable union:* $A_1, A_2, \ldots \in \bigcap \mathbf{C} \Rightarrow \forall \mathcal{X} \in \mathbf{C} \colon A_1, A_2, \ldots \in \mathcal{X} \Rightarrow \forall \mathcal{X} \in \mathbf{C} \colon \bigcup_{n=1}^{\infty} A_n \in \mathcal{X} \Rightarrow \bigcup_{n=1}^{\infty} A_n \in \bigcap \mathbf{C}$

**Definition 2.7.** *Given a collection of sets* $\mathbf{A} \subset \wp(X)$*, the* $\sigma$*-algebra* generated *by* $\mathbf{A}$*, written* $\sigma(\mathbf{A})$*, is the intersection of all* $\sigma$*-algebras containing* $\mathbf{A}$.

When $\mathbf{A}$ is a collection of pairwise disjoint sets, constructing $\sigma(\mathbf{A})$ amounts to closing $\mathbf{A}$ under countable unions; in case $\mathbf{A}$ is also finite $\sigma(\mathbf{A})$ is the power set of $\mathbf{A}$, *i.e.* $\sigma(\mathbf{A}) = \wp(\mathbf{A})$.

*Example 2.8.* The $\sigma$-algebra *generated* by $\mathbf{A} = \{[-2, 0), [0, 0], (0, 2]\}$ is
$\sigma(\mathbf{A}) = \{\emptyset, [-2, 0), [-2, 0], [0, 0], (0, 2], [0, 2], \big([-2, 0) \cup (0, 2]\big), [-2, 2]\}$.

Defining the probability of an event requires the event to be measurable, *i.e.* it occurs in the $\sigma$-algebra.

**Definition 2.9.** *A* measurable space *is a pair* $(X, \mathcal{X})$ *whereby* the sample space $X$ *is a set and* $\mathcal{X} \subseteq \wp(X)$ *is a* $\sigma$*-algebra. The elements of* $\mathcal{X}$ *are called* events *or sometimes* measurable subsets of $X$.

Probability measures are a special type of the more general measures. Naturally, every property of measures holds for probability measures.

**Definition 2.10.** *A* measure $\mu$ *on a measurable space* $(X, \mathcal{X})$ *is a function* $\mu\colon \mathcal{X} \to \mathbb{R}^+$ *that is countably additive,* i.e. *for every countable set of pairwise disjoint sets* $A_1, A_2, \ldots \in \mathcal{X}$, $\mu(\cup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} \mu(A_i)$, *and* $\mu(\emptyset) = 0$.

**Proposition 2.11.** *A* measure $\mu\colon \mathcal{X} \to \mathbb{R}^+$ *is monotone,* i.e., $A \subseteq B \Rightarrow \mu(A) \leq \mu(B)$, *whenever* $A, B \in \mathcal{X}$.

*Proof.* Let $A, B \in \mathcal{X}$ and $A \subseteq B$; then, $B = (B \setminus A) \uplus A$, implying $\mu(B) = \mu((B \setminus A) \uplus A) = \mu(B \setminus A) + \mu(A) \geq \mu(A)$ by additivity and $\mu(B \setminus A) \in \mathbb{R}^+$.

**Definition 2.12.** *A* measure space $(X, \mathcal{X}, \mu)$ *is a measurable space* $(X, \mathcal{X})$ *with a measure* $\mu$ *on it.*

**Definition 2.13.** *Given two measurable spaces* $(X, \mathcal{X}, \mu_{\mathcal{X}})$ *and* $(Y, \mathcal{Y}, \mu_{\mathcal{Y}})$, *their* product measure *is* $(X \times Y, \sigma(\mathcal{X} \times \mathcal{Y}), \mu)$ *where* $\mu(A \times B) = \mu_{\mathcal{X}}(A) \cdot \mu_{\mathcal{Y}}(B)$ *for all* $A \in \mathcal{X}$ *and* $B \in \mathcal{Y}$.

**Definition 2.14.** *A measure* $\mu$ *on a measurable space* $(X, \mathcal{X})$ *is* discrete *if its weight is on at most countably many elements,* i.e. *there exists a countable set* $A \in \mathcal{X}$ *such that* $\mu(A) = \mu(X)$, *and* continuous *if the weights of all countable sets are 0,* i.e. $\mu(A) = 0$ *for all countable sets* $A \in \mathcal{X}$.

Every measure can be uniquely represented by the sum of a discrete measure and a continuous measure.

**Definition 2.15.** *A measure* $\mu$ *on* $(X, \mathcal{X})$ *is a* probability measure *if* $\mu(X) = 1$, *and it is a* sub-probability measure *if* $\mu(X) \leq 1$.

**Proposition 2.16.** *Let* $\mu$ *be a probability measure on* $(X, \mathcal{X})$; *then,* $\mu(A) = 1 - \mu(A^{\complement})$.

*Proof.* Let $A \in \mathcal{X}$; then, $A^{\complement} \in \mathcal{X}$ by definition of $\sigma$-algebras and $A \uplus A^{\complement} = X$ by definition of complement. Since $\mu$ is countably additive, we obtain $\mu(A) + \mu(A^{\complement}) = \mu(A \uplus A^{\complement}) = \mu(X)$, and by Definition 2.15, we have $\mu(X) = 1$. Thus, $\mu(A) + \mu(A^{\complement}) = 1$, which is equal to $\mu(A) = 1 - \mu(A^{\complement})$.

*Example 2.17.* Let us define a continuous sub-probability measure $\mu_c$ over all open/closed/half-open intervals[1] over the extended reals $\mathcal{R}$,*i.e.* $\mathcal{R} = \mathbb{R} \cup \{-\infty, \infty\}$ so that it has a total weight of $1/2$, *i.e.* $\mu_c(\mathcal{R}) = 1/2$. We use a special $\sigma$-algebra, namely, the *Borel $\sigma$-algebra*, written $\mathcal{B}(\mathcal{R})$, which contains all these real intervals and may be generated by the open sets of reals; thus, $(\mathcal{R}, \mathcal{B}(\mathcal{R}), \mu_c)$. We define

[1] Let $a, b \in \mathcal{R}$. An open set $(a, b)$ of reals is defined by $(a, b) = \{x \mid x \in \mathcal{R} \wedge a < x < b\}$. A closed set $[a, b]$ of reals is defined by $[a, b] = \{x \mid x \in \mathcal{R} \wedge a \leq x \leq b\}$. A half-open set $(a, b]$ or $[a, b)$ of reals is defined as $[a, b) = \{x \mid x \in \mathcal{R} \wedge a < x \leq b\}$ or $(a, b] = \{x \mid x \in \mathcal{R} \wedge a \leq x < b\}$, respectively.

$(\mathcal{R}, \mathcal{B}(\mathcal{R}), \mu_c)$, where we may define $\mu_c$ using an integratable function $f$, as depicted in Figure 2.1a, so that $\mu_c([a,b]) = \int_a^b f(x)dx$. $f(x) = \begin{cases} \frac{1}{8}, & \text{if } x \in [-2,2] \\ 0, & \text{otherwise} \end{cases}$, *e.g.*, $\mu_c((-3,0)) = \mu_c([-3,0]) = \int_{-3}^{0} f(x)dx = 1/4$ and $\mu_c([-1,\infty)) = 3/4$.

Probability measures may be represented by probability distributions if their continuous parts are 0.

**Definition 2.18.** *A* probability space $(X, \mathcal{X}, \mu)$ *is a measure space wherein the measure* $\mu$ *is a probability measure.*

*Example 2.19.* We define a probability space $([-2,2], \mathcal{X}, \mu)$ whereby the measure $\mu = \mu_d + \mu_c$; $\mu_d$ is a discrete sub-probability measure, and $\mu_c$ is the continuous sub-probability measure from Example 2.17. Let $\mathcal{X}$ be the $\sigma$-algebra generated by $\mathbf{A} = \{[-2,0), [0,0], (0,2]\}$, as in Example 2.8, and let $\mu_d$ be defined by $\mu_d(A) = \begin{cases} \frac{1}{2} & \text{if } 0 \in A \\ 0 & \text{, otherwise.} \end{cases}$. For instance, $\mu([-2,0)) = 0.25$ and $\mu([-2,0]) = 0.75$; see Figure 2.1b for other examples. The measure $\mu$ has a total weight of 1 and is 0 for the empty set. For the $\mathbf{A}$-elements, it is $\mu([-2,0)) = 1/4$, $\mu([0,0]) = 0$, and $\mu((0,2]) = 1/4$; see Figure 2.1.

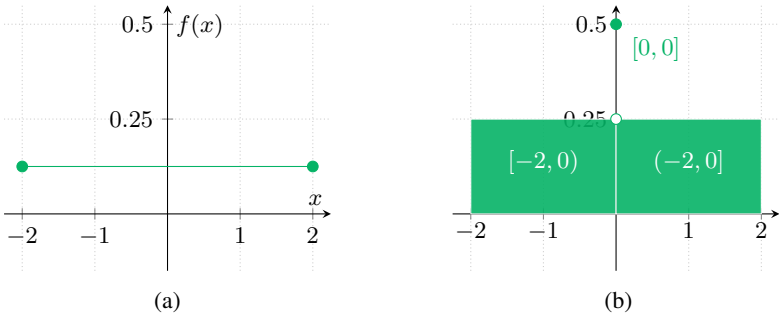



Fig. 2.1: (a) The function $f$. (b) The measure $\mu$ of $[-2,0)$,$[0,0]$, and $(0,2]$.

**Proposition 2.20.** *Let* $(X, \mathcal{X})$ *be a measurable space; then, a* probability distribution $P\colon X \to [0,1]$ *defines a (discrete) probability space* $(X, \mathcal{X}, \mu)$*, where* $\mu(A) = \sum_{x \in A} P(x)$ *whenever* $A \in \mathcal{X}$.

*Proof. Countable additive:* Since $X$ is countable (by Def. 2.1), any single set $A \in \mathcal{X}$ is countable. Thus, $\mu(A) = \sum_{x \in A} P(x)$. For a countable series of pairwise disjoint sets $A_1, A_2, \ldots \in \mathcal{X}$, we have $\mu(\cup_{i=1}^{\infty} A_i) = \sum_{x \in (\cup_{i=1}^{\infty} A_i)} P(x) = \sum_{i=1}^{\infty} \sum_{x \in A_i} P(x) = \sum_{i=1}^{\infty} \mu(A_i)$. *Measure of emptyset is 0:* $\mu(\emptyset) = \sum_{x \in \emptyset} P(x) =$

0 *Non-negative:* $\mu$ is non-negative since $P(x) \geq 0$, and a sum of non-negative numbers is non-negative. *Normalization:* $\mu(X) = \sum_{x \in X} P(x) = 1$ holds by Def. 2.1. *Discrete:* the set $X$ is countable and is in $\mathcal{X}$ (by Prop. 2.4), and $\mu(X) = 1$ (by normalization).

A probability distribution may define many measure spaces with similar behavior, that is, one for each $\sigma$-algebra.

## 2.2 Inducing probabilities

With the formal definition of probability measures and distributions in place, we are now ready to induce probability measures via a special class of functions, namely, measurable functions.

In the introduction, we saw that the probability of an output event was defined as the probability of its pre-image, and in the previous section, we saw that only measurable events may be assigned a probability. Thus, the pre-image of every measurable output event must be a measurable input event. This holds for measurable functions, which in our case are the program semantics.

**Definition 2.21.** *Let $(X, \mathcal{X})$ and $(Y, \mathcal{Y})$ be measurable spaces. A function $f\colon X \to Y$ is* measurable *if $pre_f(B) \in \mathcal{X}$ whenever $B \in \mathcal{Y}$, where the* pre-image of $f$ *$pre_f$ is defined by $pre_f(B) \triangleq \{x \in X \mid f(x) \in B\}$ and the* image $img_f(A) \triangleq \{f(x) \in Y \mid x \in A\}$. *We may write $f\colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$ when defining a measurable function.*

**Proposition 2.22.** *Let $f\colon X \to Y$ be a function, $A, B \subseteq Y$ and $\mathbf{A} \subseteq \wp(Y)$; then, the following holds.*

$$pre_f(\bigcup_{A \in \mathbf{A}} A) = \bigcup_{A \in \mathbf{A}} pre_f(A)$$
$$pre_f(\bigcap_{A \in \mathbf{A}} A) = \bigcap_{A \in \mathbf{A}} pre_f(A)$$

*Proof.* By definition of the pre-image, union and intersection, we obtain following. *union:* $x \in pre_f(\bigcup_{A \in \mathbf{A}} A) \Leftrightarrow f(x) \in \bigcup_{A \in \mathbf{A}} A \Leftrightarrow \exists A \in \mathbf{A}\colon f(x) \in A \Leftrightarrow \exists A \in \mathbf{A}\colon x \in pre_f(A) \Leftrightarrow \bigcup_{A \in \mathbf{A}} (pre_f(A))$. *Intersection:* $x \in pre_f(\bigcap_{A \in \mathbf{A}} A) \Leftrightarrow f(x) \in \bigcap_{A \in \mathbf{A}} A \Leftrightarrow \forall A \in \mathbf{A}\colon f(x) \in A \Leftrightarrow \forall A \in \mathbf{A}\colon x \in pre_f(A) \Leftrightarrow x \in \bigcap_{A \in \mathbf{A}} pre_f(A)$.

The probability of each output event $A$ is defined as the probability of $A$'s pre-image; they define a probability measure, namely, the output probability measure.

**Lemma 2.23.** *Given a probability space $(X, \mathcal{X}, \mu)$ and a measurable function $f\colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$, we define $\mu_f(A) \triangleq \mu(pre_f(A))$ whenever $A \in \mathcal{Y}$. Then, $(Y, \mathcal{Y}, \mu_f)$ is a probability space.*

*Proof.* To show that $\mu_f$ is a probability measure, we must show that it is a function $\mu_f \colon \mathcal{Y} \to \mathbb{R}^+$, where $\mu_f(\emptyset) = 0$, which is countable additive and has a total weight of 1, *i.e.* $\mu_f(Y) = 1$:

1. *a function $\mu_f \colon \mathcal{Y} \to \mathbb{R}^+$:* Since $\mu \colon \mathcal{X} \to \mathbb{R}^+$ and $pre_f \colon \mathcal{Y} \to \mathcal{X}$, their function composition $\mu \circ pre_f : \mathcal{Y} \to \mathbb{R}^+$.
2. *$\mu_f(\emptyset) = 0$:* $\mu_f(\emptyset) = \mu(pre_f(\emptyset)) = \mu(\emptyset) = 0$.
3. *countably additive:* Let $B_1, B_2, \ldots \in \mathcal{Y}$ be a countable series of pairwise disjoint sets; then, $\mu_f(\bigcup_{i=1}^{\infty} B_i) = \mu(pre_f(\bigcup_{i=1}^{\infty} B_i)) \overset{(*)}{=} \mu(\bigcup_{i=1}^{\infty} pre_f(B_i)) \overset{(**)}{=} \bigcup_{i=1}^{\infty} \mu(pre_f(B_i)) = \bigcup_{i=1}^{\infty} \mu_f(B_i)$ where $(*)$ is because the pre-images are continuous and $(**)$ because the probability measure $\mu$ is countably additive.
4. *has a total weight of 1:* $\mu_f(Y) = \mu(pre_f(Y)) = \mu(X) = 1$ where $pre_f(Y) = X$ since $f$ is total.

**Definition 2.24.** *Given two probability spaces $(X, \mathcal{X}, \mu)$ and $(Y, \mathcal{Y}, \mu_f)$ and a measurable function $f \colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$, $\mu_f$ is the* output probability measure of $f$ (with respect to $\mu$) *if $\mu_f(A) = \mu(pre_f(A)) \colon \forall A \in \mathcal{Y}$.*

For the special case whereby the input probabilities are described by a probability distribution $P \colon X \to [0, 1]$ and $f$ is a function from $X$ to another countable set $Y$, the output probabilities define a probability distribution.

**Lemma 2.25.** *Given a probability distribution $P \colon X \to [0, 1]$ and a function $f \colon X \to Y$ where $Y$ is countable, $P_f$ is a probability distribution when defined as $P_f(y) \triangleq \sum_{x \in pre_f(y)}(P(x))$ whenever $y \in Y$.*

*Proof. Countability:* by assumption *Sum to 1:* In the following, $(*)$ is obtained because the elements of $\{pre_f(y) | y \in Y \wedge pre_f(y) \neq \emptyset\}$ are pairwise disjoint since $f$ is a function, and their union equals $X$ since $f$ is total; $(**)$ is because $P$ is a probability distribution.

$$\sum_{y \in Y} P_f(y) = \sum_{y \in Y} \sum_{x \in pre_f(y)} P(x) \overset{(*)}{=} \sum_{x \in X} P(x) \overset{(**)}{=} 1.$$

**Definition 2.26.** *Given a probability distribution $P \colon X \to [0, 1]$ and a function $f \colon X \to Y$ where $Y$ is countable then $P_f$ is the* output distribution of $f$ with respect to $P$ *when defined by $P_f(y) \triangleq \sum_{x \in pre_f(y)}(P(x))$ whenever $y \in Y$.*

In the previous section, we saw that each probability distribution uniquely defines a probability measure. The input probability distribution both defines an input probability measure and infers an output probability distribution via a function; when the function is measurable, we can show that the output probability distribution defines the output probability measure (with the appropriate measurable space).

**Lemma 2.27.** *Let $f \colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$ be a measurable function, and let the probability distribution $P \colon X \to [0, 1]$ define the discrete probability space $(X, \mathcal{X}, \mu)$ and infer the output distribution $P_f$ of $f$ w. r. t. $P$, which defines the discrete probability space $(Y, \mathcal{Y}, \mu_f)$. Then, $\mu_f$ is the output probability measure of $f$ with respect to $\mu$.*

*Proof.* To show that $\mu_f$ is the output probability measure of $f$ w. r. t. $\mu$, we show that $\mu_f(A) = \mu(pre_f(A))$ and simply note that $pre_f(A) \in \mathcal{X}$ since $f$ is measurable by. To show that $\mu_f(A) = \mu(pre_f(A))$, we use Lemmas 2.20 and 2.25:

$$\mu_f(A) \overset{2.20}{=} \sum_{y\in A} P_f(y) \overset{2.25}{=} \sum_{y\in A}\sum_{x\in pre_f(y)} P(x)$$
$$= \sum_{x\in\{pre_f(y)\mid y\in A\}} P(x) = \sum_{x\in pre_f(A)} P(x) \overset{2.20}{=} \mu(pre_f(A)).$$

## 2.3 Abstract Interpretation

Abstract interpretation [28, 29] is a framework for constructing program analyses based on approximating the semantics of the language in which the program is written. The construction of an analysis involves the specification of the concrete and abstract semantics as follows.

A pair $(L, \sqsubseteq_L)$ is a partially ordered set or a poset if $L$ is a set and $\sqsubseteq_L$ is a partial order, *i.e.*, reflexive, transitive and anti-symmetric. A *complete lattice* is a poset $(L, \sqsubseteq_L)$ with a meet operator $\sqcap$ and a join operator $\sqcup$ in which each subset $S \subseteq L$ has a least upper bound $\bigsqcup S$ and greatest lower bound $\bigsqcap S$. Furthermore, $\bot = \bigsqcup \emptyset = \bigsqcap L$ is *the least element*, and $\top = \bigsqcap \emptyset = \bigsqcup L$ is *the greatest element*.

The concrete semantics of a program is defined over a *concrete domain*, *i.e.*, a complete lattice $(L, \sqsubseteq_L)$, and the abstract semantics of a program is defined over an *abstract domain*, *i.e.*, a complete lattice $(M, \sqsubseteq_M)$. The program semantics is described by a *concrete transformer*, *i.e.*, a monotone function $f\colon L \to L$, and by an *abstract transformer*, *i.e.*, a monotone function $g\colon M \to M$. The two domains are connected by a pair of functions: *abstraction* $\alpha\colon L \to M$ and *concretization* $\gamma\colon M \to L$. The pair $(\alpha, \gamma)$ is a *Galois connection* if $\forall l \in L$ and $\forall m \in M$ we have that $\alpha(l) \sqsubseteq_{\mathrm{M}} M \Leftrightarrow l \sqsubseteq_{\mathrm{L}} \gamma(M)$. An abstract transformer $g$ is an *abstraction* of a concrete transformer $f$ if $(f \circ \gamma)(m) \sqsubseteq_L (\gamma \circ g)(m)$ whenever $m \in M$.

An archetypal example of an abstract interpretation is interval analysis. Here, the concrete domain is $\wp(\mathbb{R})$ with set inclusion as partial order, and the abstract domain is the set of real intervals $\mathcal{I} = \bot \cup \{(x, y) \mid x, y \in \mathcal{R}, x < y\}$ with containment as partial order, *i.e.*, $(x_1, y_1) \sqsubseteq (x_2, y_2)$ if and only if $x_2 \leq x_1 \wedge y_1 \leq y_2$. The abstraction $\alpha(\emptyset) = \bot$; otherwise, $\alpha(S) = (\min(S), \max(S))$ with $\min(\mathbb{R}) = -\infty$ and $\max(\mathbb{R}) = \infty$, and the concretization $\gamma((x, y)) = \{z \mid x \leq z \leq y\}$. This defines a Galois connection. The domains are extended to tuples of elements of $\mathbb{R}^n$ and $\mathcal{I}^n$, respectively. The abstract transformer on intervals is constructed systematically from the concrete operations. For example, we derive an abstract interval operation $+^\sharp\colon \mathcal{I} \times \mathcal{I} \to \mathcal{I}$ from its concrete counterpart $+ : \wp(\mathbb{R}) \times \wp(\mathbb{R}) \to \wp(\mathbb{R})$; that is, the concrete addition $A_1 + A_2 = \{a_1 + a_2 \mid a_1 \in A_1, a_2 \in A_2\}$ defines an abstract ditto $\alpha(A_1) +^\sharp \alpha(A_2) = \alpha(A_1 + A_2) = (\min(A_1 + A_2), \max(A_1 + A_2)) = (\min(A_1) + \min(A_2), \max(A_1) + \max(A_2))$, *i.e.* $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$.

# 3

# Discrete probabilistic output analysis

In this chapter, we focus on deterministic programs over countable input and represent the discrete input and output measures by probability distributions. These probability distributions may be parameterized, *e.g.* a uniform probability distribution between $0$ and a natural number $n$. For a given output event $A$, we express its probability (using Definition 2.26) as the sum of probabilities of all inputs that the program relates to the output. Then, we infer an upper probability bound by manipulating and transforming this sum-expression into a closed-form expression. A closed-form expression is computable (in a finite number of steps) and may not contain infinite sums or products; we ensure this by defining closed-form expressions to be expressions without summations, products or recursive expressions. The analysis draws on recurrence equation methods known from resource analysis [109]. The analysis yields upper probability bounds.

*Notation and Terminology.* An important difference in notation is the use of "over approximation" and "under approximation"; in the following article, we use "an over approximation" when referring to an upper bound of a function and "an under approximation" when referring to a lower bound of a function; see for instance Definition 3.4. Where we previously used $f^\sharp$ and $f^\flat$ to refer to an upper and lower bound, respectively, we will here use $\overline{f}$ and $\underline{f}$, respectively.

*Foreword.* The remainder of this chapter – except for the afterword (Section 3.10) – is published with minor corrections in article [118] M. Rosendahl, M. H. Kirkeby. Probabilistic Output Analysis by Program Manipulation. *Electronic Proceedings in Theoretical Computer Science*, 194(318337):110–124, 2015.

**Abstract.** The aim of a probabilistic output analysis is to derive a probability distribution of possible output values for a program from a probability distribution of its input. We present a method for performing the output analysis based on program transformation techniques. The method generates a probability function as a possibly uncomputable expression and transforms that into a closed-form expression. The probability functions are viewed as programs in

a separate language in which they may be analysed, transformed, and approximated. We focus on programs where the possible input follows a known probability distribution. Tests in programs are not assumed to satisfy the Markov property of having fixed branching probabilities independently of previous history.

## 3.1 Introduction

The aim of a probabilistic output analysis (POA) is to derive a probability distribution for output values from a probability distribution for input to a program. The internal properties of a program can also be analysed in this manner by instrumenting programs with step counters for complexity analysis [116] or energy consumption measures [86].

When analysing energy consumption, probability distributions may provide more useful information than boundaries. Wierman et al. state that *"global energy consumption is affected by the average case, rather than the worst case"* [144]. In addition, in scheduling, *"an accurate measurement of a tasks average-case execution time (ACET) can assist in the calculation of more appropriate deadlines"* [62]. For a subset of programs, a precise average case execution time can be found using static analysis [51, 57, 124]. In some cases, the POA delivers not only an accurate output average but a more descriptive accurate output distribution. In other cases, the POA must over-approximate the probability distribution, and the expected value (average case result) will be approximated safely as a range. Another application area for POA is in temperature management, where worst-case bounds are important [125]. Because POA return distributions, it can be used to calculate the probability of energy consumptions above a certain limit, thereby indicating the risk of over-heating.

The main contribution in this paper is to present a technique for probabilistic analysis whereby the analysis is seen as a program-to-program translation. This means that the transformation to closed form is a source code program transformation problem and not specific to the analysis. Any necessary approximation in the analysis is also performed at the source code level. The technique also makes it possible to balance the precision of the analysis against the brevity of the result.

The method in this paper is inspired by the techniques used in automatic complexity analysis. Wegbreit's Metric system [141] laid the ground work for many later systems with an aim of deriving best-, worst- and average-case complexity measures. Later works in this area have focused on worst-case complexity [8, 87, 116] with advanced systems that can analyse realistic programs. The approach in this paper uses an approach similar to [116] in that we derive the probability function without approximations and introduce approximations only in the last phase. We transform the original program into a program that computes the probability distribution. The intermediate stage is then a potential subject of further analysis based on abstract interpretation. This program can be analysed, transformed, and approximated. It is thus an alternative to deriving cost relations directly from the program [8, 87] or expressing costs as abstract values in a semantics for the language.

As with automatic complexity analysis, the aim of probabilistic output analysis is to express the result as a parameterized expression. The time complexity of a program should be given as a closed-form expression in the input size, and for probabilistic output analysis, the aim is to express the probability of the output values of the program as a function in output values and input size or range. As a small example, let us consider the addition `add` of two independent integer values `x` and `y` evenly distributed from 1 to $n$. It is a tail-recursive program where the output distribution is well-known to be a pyramid-shaped distribution. The input probability distributions should also be expressed in the language, as they are part of the transformational approach to obtain the output probability distribution.

```
add(x,y) = if(x=0) then y else add(x-1,y+1)
px(x,n) = if(x >= 1 and x <= n) then 1/n else 0
py(y,n) = if(y >= 1 and y <= n) then 1/n else 0
```

Our probabilistic output analysis returns a function describing the probability distribution of the output:

```
padd(z,n) =
  1/(n*n)*max(min(n,z-1) - max(1,z-n) + 1,0)
```

The analysis can also be used for more complex input distributions and programs, but it will not always be able to reduce it to closed form. If this is not possible, we will approximate the distribution and thus obtain an over-approximation of the extreme cases and a range for the expected value. If the input values are not independent, we can specify a joint distribution for the values. The values do not have to be restricted to a finite range; however, for infinite ranges, the distribution would converge to zero in the limit.

## 3.2 Probability distributions

The analysis presented here is based on using a discrete set of values for input and output. The set will be finite or countable, and we will use discrete probability distributions. It is also possible to use an uncountable set of values or a combination of discrete and continuous random variables if one uses cumulative probability measures in the analysis. This will be further discussed later in the paper.

We consider the input to a program as a discrete random variable; the input probability distribution is then a probability measure that assigns a value between 0 and 1 to an event of input having a given value. This is also often referred to as the *probability mass function* in the discrete case, and in the continuous case, one will use a *probability density function*. We will use the phrase *probability distribution* to denote mappings from single values (input or output) to a probability or number between 0 and 1, and we will use upper case $P$ letters to denote such functions.

**Definition 3.1 (probability distribution).** *For a countable set $X$, a probability distribution over $X$ is a mapping $P_X : X \to [0, 1]$, where*

$$\sum_{x \in X} P_X(x) = 1$$

We define the output probability distribution for a program $\mathtt{p}$ in a forward manner. It is the *weight* or sum of all probabilities of input values where the program returns the desired value $z$ as output.

**Definition 3.2 (output probability).** *Given a program,* $\mathtt{p} \colon X \to Z$*, and a probability distribution over the input,* $P_X$*, the probability distribution over the output,* $P_{\mathtt{p}} \colon Z \to [0,1]$*, is defined as*

$$P_{\mathtt{p}}(z) = \sum_{x \in X \wedge \mathtt{p}(x) = z} P_X(x)$$

Note that Kozen also uses a similar forward definition [83], whereas Monniaux constructs the inverse and expresses the relationship in a backward style [95].

**Lemma 3.3.** *The output probability distribution,* $P_{\mathtt{p}} \colon Z \to [0,1]$*, satisfies*

$$0 \leq \sum_{z \in Z} P_{\mathtt{p}}(z) \leq 1$$

The program may not terminate for all input, and thus, the sum may be less than one. If we expand the domain $Z$ with an element to denote non-termination, $Z_\perp$, the total weight of the output distribution $P_{\mathtt{p}}$ would be 1.

**Approximations of probability distributions.** The output analysis cannot necessarily derive the precise probability distribution. Various approaches to approximations of probability distributions have been proposed and can be interpreted as *imprecise probabilities* [4, 37, 46]. Dempster-Shafer structures [14, 61] and P-boxes [48] can be used to capture and propagate uncertainties of probability distributions. There are several results on extending arithmetic operations to probability distributions for both known dependencies between random variables and when the dependency is unknown or only partially known [16, 18, 113, 136, 145]. Algorithms for lifting basic operations on numbers to basic operations on probability distributions can be used as abstractions in static analysis based on abstract interpretation. Our approach defines an upper bound of the probability distribution. Later, we shall discuss this representation with P-boxes, *i.e.* a pair of upper and lower cumulative probability distributions.

**Definition 3.4 (over- and under-approximation).** *Let* $P_{\mathtt{p}} \colon Z \to [0,1]$ *be a distribution and* $\overline{P}_{\mathtt{p}}, \underline{P}_{\mathtt{p}} \colon Z \to [0,1]$ *be functions; then,*

- $\overline{P}_{\mathtt{p}}$ *is an* over-approximation of $P_{\mathtt{p}}$ *if* $P_{\mathtt{p}}(z) \leq \overline{P}_{\mathtt{p}}(z) \leq 1$*, and*
- $\underline{P}_{\mathtt{p}}$ *is an* under-approximation of $P_{\mathtt{p}}$ *if* $0 \leq \underline{P}_{\mathtt{p}}(z) \leq P_{\mathtt{p}}(z)$.

The aim of the probabilistic output analysis is to derive as tight approximations $\underline{P}$ and $\overline{P}$ as possible.

**Lemma 3.5.** *Given an over- and under-approximation* $\overline{P}_{\mathrm{p}}, \underline{P}_{\mathrm{p}} \colon Z \to [0,1]$ *of* $P_{\mathrm{p}}$*, their total weights will be bounded as*

$$0 \leq \sum_{z \in Z} \underline{P}_{\mathrm{p}}(z) \leq 1 \qquad \textit{and respectively} \qquad 0 \leq \sum_{z \in Z} \overline{P}_{\mathrm{p}}(z) \leq \infty$$

When $\underline{P}_{\mathrm{p}} = \overline{P}_{\mathrm{p}}$, the total weight for each function will be equal to the total weight of $P_{\mathrm{p}}$ according to definition 3.4. For terminating programs, the total weight is 1.

**Expected value.** Provided that the output from the program is numerical, one may be interested in the average output value of the program. In this context, this is *the expected value* of the output distribution. If the program does not terminate for all input, it is not clear how to define the expected value because non-termination may indicate a possibly infinite output value. As part of the further analysis, we need a guarantee that the program terminates. If the weight of $\underline{P}_{\mathrm{p}}$ is 1, then we know that the program terminates for all possible input (*i.e.* input with probability greater than zero).

**Lemma 3.6.** *Let* $\underline{P}_{\mathrm{p}} \colon Z \to [0,1]$ *be an under-approximation of a probability distribution* $P_{\mathrm{p}} \colon Z \to [0,1]$*; then,*

$$\sum_{z \in Z} \underline{P}_{\mathrm{p}}(z) = 1 \Rightarrow \sum_{z \in Z} P_{\mathrm{p}}(z) = 1$$

The expected value of the output distribution is defined as the weighted average of the distribution.

**Definition 3.7 (expected value).** *The expected value* $E_{\mathrm{p}}$ *of the output distribution* $P_{\mathrm{p}} \colon Z \to [0,1]$ *is defined as*

$$E_{\mathrm{p}} = \sum_{z \in Z} z \cdot P_{\mathrm{p}}(z)$$

If we cannot analyse the program precisely, we can use the over-approximation to compute an interval for the expected value. We cannot use the approximation $\overline{P}_{\mathrm{p}}$ directly, as its weight is not necessarily 1. Using $\overline{P}_{\mathrm{p}}$, we can create two new probability distributions, each with a total weight of 1. One distribution favors the lower values, and one distribution favors the higher values. These two distributions can then be used to calculate a lower and upper bound for the expected values.

**Definition 3.8 (expected value interval).** *For an over-approximation of a probability distribution* $\overline{P}_{\mathrm{p}} \colon Z \to [0,1]$*, we first define* over- and under-cumulative functions*, namely,* $F^{\uparrow}_{\mathrm{p}}, F^{\downarrow}_{\mathrm{p}} \colon Z \to [0,1]$.

$$F^{\uparrow}_{\mathrm{p}}(z) = \min(\sum_{v \leq z} \overline{P}_{\mathrm{p}}(v), 1)$$

$$F^{\downarrow}_{\mathrm{p}}(z) = \max(1 - \sum_{v \geq z} \overline{P}_{\mathrm{p}}(v), 0)$$

*Using these, we define over- and under-expected value ($E^{\uparrow}_{\mathtt{p}}$ and $E^{\downarrow}_{\mathtt{p}}$).*

$$E^{\downarrow}_{\mathtt{p}} = \sum_z z \cdot (F^{\uparrow}_{\mathtt{p}}(z) - F^{\uparrow}_{\mathtt{p}}(dec(z)))$$
$$E^{\uparrow}_{\mathtt{p}} = \sum_z z \cdot (F^{\downarrow}_{\mathtt{p}}(z) - F^{\downarrow}_{\mathtt{p}}(dec(z)))$$

*where* $dec(z) = \max\{v \in Z \mid v < z\}$.

Notice that an expected value based on an over-approximation of the cumulative probability gives an under-approximation of the expected value. If the output space $Z$ is integers, then the *dec* function will simply subtract one from its argument.

**Lemma 3.9 (expected value interval).** *For a terminating program, the expected value can be approximated by an interval from the over-approximation of the probability distribution.*

$$E^{\downarrow}_{\mathtt{p}} \leq E_{\mathtt{p}} \leq E^{\uparrow}_{\mathtt{p}}$$

**Externalise resource usage.** The output analysis can be used to analyse internal properties of the program provided these properties are externalised. As in automatic worst-case complexity analysis [116], this may be done by instrumenting the program with step counting information. Similarly, we might instrument programs with energy consumption based on low-level energy models for operations [86] to be able to analyse programs in terms of average energy consumption.

An operational or denotational semantics of a simple first-order functional programming language would normally describe programs as mappings of input values to output values. The time, space and energy required to perform the computation would normally not be part of the semantics. The simplest form of a resource analysis is to count the number of basic operations that a computation would require. An automatic complexity analysis [116] is then based on a semantics that has been extended (or instrumented) with step-counting information so that the meaning of a program is a mapping of input values to a tuple of the output and the number of steps. If we write this semantics as an interpreter in the source language, we can convert a program into a step-counting version of the program by partial evaluation. In this way, the complexity analysis has been transformed into an output analysis of the program. The aim of the complexity analysis is then to generate an under-approximation of the (second component of the) possible output as a function of the size of the possible input. If we instrument the semantics with other types of resource information, we can analyse programs with respect to these properties. Some automatic complexity analysis systems are based on translating programs into cost relations [8] or cost equations [87]. These approaches are then used as approximations of an instrumented semantics that captures the cost of computations.

**The challenge of approximation.** The analysis of probabilistic behaviour introduces some new challenges compared to worst-case analysis. It is well known that a

function of expected values is not necessarily the same as the expected value of the function. There are a number of other potential pitfalls when making approximations in a probabilistic setting. One might assume that conditions in a program can be assigned a fixed probability of being true independently of previous execution paths in the program. One might also assume that variables have independent probability distributions. An unfortunate effect of using independence as an approximation is that it tends to under approximate the extreme cases. In a throw of two dice, the sum being 12 has probability $1/36$ if we can assume independence. If (by some magic) the dice always showed the same face, the probability increases to $1/6$. The situation is well-known in the insurance industry and for financial risk management (valuation of derivatives), where one may want to over-approximate the risk of extreme events when events are not guaranteed to be independent. One approach to handle such situations is the use of copulas [17] and comonotonicity of probability measures [38].

## 3.3 Transformation-Based Analysis

Our analysis is based on a small first-order functional language with a simple recursive structure. The first step of the analysis is to translate programs into a new language of probability distribution programs. We will then use analysis and transformation techniques to transform the probability distributions into closed form. Failing that, we may approximate the distribution with an upper and lower approximation ($\overline{P}$ and $\underline{P}$).

Programs have the form of a collection of functions

$$\begin{aligned} &f_1(x_1, \ldots, x_n) = e_1 \\ &\vdots \\ &f_n(x_1, \ldots, x_n) = e_n \end{aligned}$$

The language uses a base set $D$ of values for simple expressions, and functions in a program denote mappings from tuples of values to values $D^* \to D$.

The base set of values will not be further restricted here nor do we specify the exact set of basic operations in the language. The first function in the program is called externally, and for that function, we have an input probability distribution $P_x$ specified as a symbolic expression $e_x$.

There are two forms of functions: non-recursive and recursive functions.

$$f(x_1, \ldots, x_n) = \textbf{ if } (b(x_1, \ldots, x_n)) \textbf{ then } g(x_1, \ldots, x_n) \textbf{ else } f(e_1 \ldots, e_n)$$

Non-recursive functions have right-hand sides that are built from simple operations, conditional expressions, and function calls to non-recursive and recursive functions.

### 3.3.1 Probability distribution program

When constructing the probability distribution program, in its raw form, we use two new language constructs: Sums over the (possibly) infinite set of all input values in $D^*$ and a constraint function $C$. The constraint function eases the handling of boundaries and is defined as

$$C(condition) = \begin{cases} 1 & \text{if } condition = true \\ 0 & \text{otherwise} \end{cases}$$

This definition is related to the indicator function [101] or characteristic function for memberships of sets. We also extend the language with a finite product construction that will be used for unfolding the simple tail recursions.

The output distribution program is expressed in a language similar to the original program but extended with two classes of functions: the original functions of type $D^* \to D$ and probability functions of type $D^* \to [0, 1]$. One of these functions will be the output distribution function of type $D \to [0, 1]$.

The output probability distribution program is defined based on the program that we want to analyse and a probability distribution over its input. The input distribution is expressed as a program function

$$P_x(x_1, \ldots, x_n) = e_x$$

from input to their probabilities. If the input-arguments are independent the function can be written as a product of the probability distribution of each argument

$$P_x(x_1, \ldots, x_n) = P_{x1}(x_1) \cdots P_{xn}(x_n)$$

The raw form of the probability distribution program is defined as follows. The output distribution program $P_f$ takes an output and returns the sum of all probabilities of those inputs that the original program maps to the specified output. The output probability distribution program $P_f$ of the program function $f_1$ is defined as follows.

$$\begin{aligned} &P_f(z) = \sum_{x_1} \cdots \sum_{x_n} P_x(x_1, \ldots, x_n) \cdot C(z = f_1(x_1, \ldots, x_n)) \\ &P_x(x_1, \ldots, x_n) = e_x \\ &f_1(x_1, \ldots, x_n) = e_1 \\ &\vdots \\ &f_n(x_1, \ldots, x_n) = e_n \end{aligned}$$

We interpret a probability distribution program as a program that can be transformed and analysed. The second phase is to unfold function calls, and the following phase is to attempt to remove the infinite summations. The aim of the subsequent transformation stages is to remove the infinite summations from the program and, in the process, the functions from the original program.

## 3.4 Method

The analysis technique can be divided into two parts: the unfolding and the symbolic summation. Unfolding replaces the recursive structures, function calls and conditional expressions with semantically equivalent expressions that have a straightforward mathematical semantics.

### 3.4.1 Unfolding

In this phase, we unfold function calls in the program. We will introduce the central transformation rules for unfolding calls into functions in the original program based on the syntactical structure.

**Function calls.** Simple calls to functions can be unfolded directly. Calls to recursive functions can be composed, but each call can be analysed separately by constructing a joint input distribution function to the call. For such function calls, we rewrite the program as follows:

$$\begin{aligned}&\sum_{x_1}\cdots\sum_{x_n} P(x_1,\ldots,x_n)\cdot C(z=f(e_1,\ldots,e_n))\\&=\sum_{u_1}\cdots\sum_{u_n} P_c(u_1,\ldots,u_n)\cdot C(z=f(u_1,\ldots,u_n))\end{aligned}$$

The rewritten expression requires a new function $P_c$:

$$P_c(u_1,\ldots,u_n)=\sum_{x_1}\cdots\sum_{x_n} P(x_1,\ldots,x_n)\cdot C(u_1=e_1)\cdots C(u_n=e_n)$$

Since we assume that the programs do not have unrestricted recursion, we will only generate a bounded number of extra probability functions.

**Conditional expressions.** For conditional expressions, we use the following rule:

$$\begin{aligned}&\sum_{x_1}\cdots\sum_{x_n} P(x_1,\ldots,x_n)\cdot\\&\quad C(z=\ \textbf{if}\ (b(x_1,\ldots,x_n))\ \textbf{then}\ f(x_1,\ldots,x_n)\ \textbf{else}\ g(x_1,\ldots,x_n))\\&=\sum_{x_1}\cdots\sum_{x_n} P(x_1,\ldots,x_n)\cdot\\&\quad (C(b(x_1,\ldots,x_n))\cdot c(z=f(x_1,\ldots,x_n))+\\&\quad C(\neg b(x_1,\ldots,x_n))\cdot c(z=g(x_1,\ldots,x_n))\,)\end{aligned}$$

**Unfolding recursion.** For the simple tail recursion, we collect the probability of a given result being returned after any number of recursive calls. The condition may never evaluate to true for a certain input (non-termination), and in that situation, the sum of output probabilities will be less than 1.

The recursive functions have the form

$$f(x_1, \ldots, x_n) = \textbf{ if } (b(x_1, \ldots, x_n)) \textbf{ then } g(x_1, \ldots, x_n) \textbf{ else } f(e_1 \ldots, e_n)$$

and they should be analysed for all input probability distributions that we detect at calls to these functions.

The transformation for the recursive form is

$$\begin{aligned}
&\sum_{x_1} \cdots \sum_{x_n} P(x_1, \ldots, x_n) \cdot \\
&\quad C\,(z = \textbf{ if } (b(x_1, \ldots, x_n)) \textbf{ then } g(x_1, \ldots, x_n) \textbf{ else } f(e_1 \ldots, e_n)) \\
&= \sum_{x_1} \cdots \sum_{x_n} P(x_1, \ldots, x_n) \cdot \\
&\quad \sum_{i=0}^{\infty} \prod_{j=0}^{i-1} C(\neg b(h(j, x_1, \ldots, x_n))) \cdot C(b(h(i, x_1, \ldots, x_n))) \\
&\qquad\qquad \cdot C(z = g(h(i, x_1, \ldots, x_n)))
\end{aligned}$$

where

$$h(i, x_1, \ldots, x_n) = \textbf{ if } (i = 0) \textbf{ then } \langle x_1, \ldots, x_n \rangle \textbf{ else } h(i-1, e_1, \ldots, e_n)$$

In the transformed expression, we introduce two variables: $i$, which represents the number of recursive calls, and $j$, which represents all previous recursions for the $i$ under investigation (when $i$ is 0, the term $\prod_{j=0}^{i-1} C(\neg b(h(j, x_1, \ldots, x_n)))$ evaluates to 1). The new function $h(i, x_1, \ldots, x_n)$ describes the evaluation of the expressions $\langle e_1, \ldots, e_n \rangle$, $i$ times. Only when the $i$th condition is *true* and all previous conditions are *false* can the expression evaluate to a probability of greater than 0.

### 3.4.2 Symbolic summation

In the previous phase, we unfolded calls to functions in the original program. The next phase uses algebraic transformation techniques to remove summations. The transformations that we use are similar to those used in worst case execution-time systems for solving recurrence equations [87, 117] or symbolic summation techniques in loop-bound computations [80]. Some of the central transformation rules that we use in this phase are listed below. In the following transformations, the expressions $e_1$ and $e_2$ are assumed not to contain the summation variable $x$.

$$\sum_x C(x = e_1) \cdot f(x) = f(e_1)$$

$$\sum_x C(e_1 \leq x \leq e_2) = (e_2 - e_1 + 1) \cdot C(e_1 \leq e_2)$$

$$\sum_x x \cdot C(e_1 \leq x \leq e_2) = \left( \frac{e_2 \cdot (e_2 + 1)}{2} - \frac{e_1 \cdot (e_1 - 1)}{2} \right) \cdot C(e_1 \leq e_2)$$

One could also use computer algebra systems in the reduction process, but some of the rules are quite specific to how we handle the boundaries of summations with the special constraint function. There are a number of rules for combining products of constraint functions and for splitting intervals into separate expressions.

$$C(e_1 \leq x \leq e_2) \cdot C(e_3 \leq x \leq e_4) = C(\max(e_1, e_3) \leq x \leq \min(e_2, e_4))$$

$$C(\max(e_1, e_2) \leq e_3) = C(e_1 > e_2) \cdot C(e_1 \leq e_3) + C(e_1 \leq e_2) \cdot C(e_2 \leq e_3)$$

There are similar rules for removing the minimum function and for isolating variables in constraints.

There are also rules for the symbolic summation of certain infinite summations. If $a$ is an expression whereby $0 < a < 1$, then we can simplify the expression as follows:

$$\sum_x C(x \geq 0) \cdot a^x = \frac{1}{(1-a)}$$

$$\sum_x C(x \geq 0) \cdot x \cdot a^x = \frac{1}{(1-a)^2} - \frac{1}{(1-a)}$$

This rule is useful when some of the input to the program follows a geometric distribution.

$$P_x(x, n) = C(x \geq 0) \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^x$$

## 3.5 Examples

In the following, we calculate two examples using the rules from before; the examples contain a conditional branch and a recursion.

**Max example.** As a small example, let us look at the simple non-recursive program `max`, which, given two values, returns the largest value This program is chosen because it only makes use of the symbolic summation rules and because the output follows a non-uniform distribution even if the input variables are uniformly distributed. The program is defined as

```
max(x,y) = if (x>y) then x else y
```

The input probabilities are independent; each is a uniform distribution from 1 to $n$ and can be defined as

$$P_x(x) = \frac{1}{n} \cdot C(1 \leq x \leq n) \quad \text{and}$$

$$P_y(y) = \frac{1}{n} \cdot C(1 \leq y \leq n)$$

The following example uses the conditional transformation rule and the symbolic summation rules.

$$
\begin{aligned}
P_{\texttt{max}}(z) &= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z = \texttt{if } (x > y) \texttt{ then } x \texttt{ else } y) \\
&= \frac{1}{n^2} \cdot \Big( \sum_y \big( C(1 \le z \le n) \cdot C(1 \le y \le n) \cdot C(y \le (z-1)) \big) \\
&\qquad + \sum_x \big( C(1 \le x \le n) \cdot C(1 \le z \le n) \cdot C(x \le z) \big) \Big) \\
&= \frac{1}{n^2} \cdot (2z - 1) \cdot C(1 \le z \le n)
\end{aligned}
$$

**Add example.** The recursive addition function was used as an example in the introduction. We shall see how the original program is inserted into the probability formula, expanded and reduced to a closed-form function expressing the probability distribution for the output. Recall the program

```
add(x,y) = if(x=0) then y else add(x-1,y+1)
```

and that we assume independence between the input variables; for the sake of simplicity, we let both input variables `x` and `y` follow a uniform distribution from 1 to a number $n$.

$$
\begin{aligned}
P_{\texttt{add}}(z) &= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \\
&\sum_{i=0} \prod_{j=0}^{i-1} C(\neg b(h(j,x,y))) \cdot C(b(h(i,x,y))) \cdot C(z = g(h(i,x,y))
\end{aligned}
$$

where

$$
\begin{aligned}
b(x,y) &= x = 0 \\
g(x,y) &= y \\
h(i,x,y) &= \textbf{if } (i = 0) \textbf{ then } \langle x, y \rangle \textbf{ else } h(i-1, x-1, y+1) \\
&= \langle x - i, y + i \rangle
\end{aligned}
$$

$$
\begin{aligned}
P_{\texttt{add}}(z) &= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \\
&\qquad \sum_{i=0}^{i-1} \prod_{j=0} C(\neg(x - j = 0)) \cdot C(x - i = 0) \cdot C(z = y + i) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \sum_{i=0} C(x = i) \cdot C(z = y + i) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z = y + x) \\
&= \sum_y \frac{1}{n} \cdot C(z - n \leq y \leq z - 1) \cdot \frac{1}{n} \cdot C(1 \leq y \leq n)\big) \\
&= \frac{1}{n^2} \cdot \max(\min(n, z - 1) - \max(1, z - n) + 1, 0) \\
&= \frac{1}{n^2} \cdot \big(C(n < z \leq 2n) \cdot (2n - z + 1) \\
&\qquad + C(1 \leq z \leq n) \cdot (z - 1)\big)
\end{aligned}
$$

### 3.5.1 Expected value

If we have derived a probability program, we may also derive an expression that computes the expected value of the distribution.

$$E_{\mathtt{p}} = \sum_x x \cdot P_{\mathtt{p}}(x)$$

For the `add` program, this gives

$$E_{\texttt{add}} = \sum_{z=1}^{n} z \cdot \frac{1}{n^2} \cdot (z - 1) + \sum_{z=n+1}^{2n} z \cdot \frac{1}{n^2} \cdot (2n - z + 1)$$

which, of course, can be reduced further.

## 3.6 Non-primitive Types

In the approach, we have stated that the base domain is a countable set and not necessarily simply numbers; more complicated types, such as lists, can be used. Our only requirement was that we must be able to define a probability distribution for values in the domain. In the following, we will show how a list behaves for a simple `member` program and indicate how conditions may affect the probability distribution for the list.

In the following, we assume that the lists are non-empty and of length $k$. Furthermore, we assume that the elements are independent of each other and that each

element is uniformly distributed over the interval 1 to $n$. The following probability function describes this.

$$P_L(L) = \frac{1}{n^k} \cdot C(\mathit{length}(L) = k \wedge \forall j : 0 \leq j \leq k-1 \wedge 1 \leq hd(tl^j(L)) \leq n)$$

It assigns the probability $1/n^k$ to any list of length $k$ where all elements are uniformly distributed in the interval from 1 to $n$.

If we consider the member function for non-empty lists, it can be written as

```
member(X,L) =
  if(tl(L)=[] || hd(L)=X) then hd(L)=X
  else member(X,tl(L))
```

The function will follow the pattern of recursion as described earlier, and the output probability function for the member function is then

$$P_{\texttt{member}}(z) = \sum_X \sum_L P_X(X) \cdot P_L(L) \cdot C(z = \texttt{member}(X, L))$$

We can apply the transformation rules to simplify the expression into closed form. The lists were here assumed to possibly contain repeating elements. We could also use a different probability distribution to restrict lists to non-repeating lists of values. This restriction is made by Wegbreit [141] in his examples, where the probability is derived as $1 - (1 - (1/n)^k)$, which is the correct result for repeating lists of values.

*Conditional expressions and lists.* Sometimes, the conditional expressions influence the possible lists and thereby the probability distribution. Wegbreit's technique is valid for programs where one can safely assume the Markov property (that the probabilities of conditions are fixed). Wegbreit observes that this is not always true even in other cases, even simple cases, e.g., in nested conditionals where the outcome of the first condition influences the probability of the outcome for the subsequent condition. This phenomena is sometimes referred to as *gain of knowledge*. For instance, consider the following `union` function for two repeating lists `L1` and `L2`.

```
fun union L1 L2 =
 if L1 = [] then y else
 if member (hd L1) L2  (* Gain of knowledge *)
    then union (tl L1) L2 else
         (hd L1)::(union (tl L1) L2)
```

In `union`, the gain of knowledge occurs in the test '`member (hd L1) L2`', which checks whether the head of `L1` is a member of `L2`; the outcome of this test affects the probability of its next outcome, that is, if the head of `L1` is not in `L2`, the likelihood of the next element of `L1` not being in `L2` increases slightly.

Both of the following formulas[1] express the probability of the next element of `L1` not being in `L2`; however, formula (3.2) expresses the probability when we know that the first element is not a part of the list.

[1] The formulas assume that the lists have a length of greater than 1.

$$P(hd(tl(\texttt{L1})) \notin \texttt{L2}) = \frac{(n-1)^k}{n^k} \tag{3.1}$$

$$P(hd(tl(\texttt{L1})) \notin \texttt{L2} \mid hd(\texttt{L1}) \notin \texttt{L2}) = \frac{1}{n} \cdot \frac{(n-1)^k}{(n-1)^k} + \frac{n-1}{n} \cdot \frac{(n-2)^k}{(n-1)^k} \tag{3.2}$$

The maximum difference is $50\%$ and occurs when there are only two different elements $n = 2$ and the length $k$ is going towards $\infty$. In that specific case, the check provides us knowledge on whether `L2` contains one or two types of element.

## 3.7 Approximation Techniques

The probability distribution program expresses the probability distribution for output values. Our aim is to transform it into a closed form, but this may not always be possible. Failing that, we can instead use approximation techniques to obtain an upper bound for the probability distribution. We have referred to this as the over-approximation of the probability distribution: $\overline{P}$. The techniques that we may apply here are similar to automatic worst-case complexity analysis [116], where the aim is to obtain a closed-form expression for the complexity of programs but, failing that, where we may obtain an over-approximation.

**Cumulative distribution functions.** Cumulative probabilities will in some cases be more useful and expressive than probability distributions. Cumulative probabilities can be used in both the discrete and continuous case, and in some cases, approximations can be described more precisely using cumulative probabilities than with ordinary distributions. It tends, however, to be more complex to reduce to closed forms and thus may require coarser approximations. The bounding of a cumulative distribution was introduced by Ferson [48] as a P-box and can be used to describe imprecise probability distributions.

**Definition 3.10 (cumulative distribution).** *Given a program output probability distribution, $P_{\texttt{p}}(z)$, the cumulative program output probability distribution, $F_{\texttt{p}}(z)$, is defined as*

$$F_{\texttt{p}}(z) = \sum_{w \leq z} P_{\texttt{p}}(w)$$

**Definition 3.11 (over- and under-approximation).** *Given a cumulative output probability of a program `P`, $F_{\texttt{p}}$, the over-approximation, $\overline{F}_{\texttt{p}}$, and the under-approximation, $\underline{F}_{\texttt{p}}$, are defined as*

$$\overline{F}_{\texttt{p}} : \forall z. F_{\texttt{p}}(z) \leq \overline{F}_{\texttt{p}}(z) \qquad\qquad \underline{F}_{\texttt{p}} : \forall z. \underline{F}_{\texttt{p}}(z) \leq F_{\texttt{p}}(z)$$

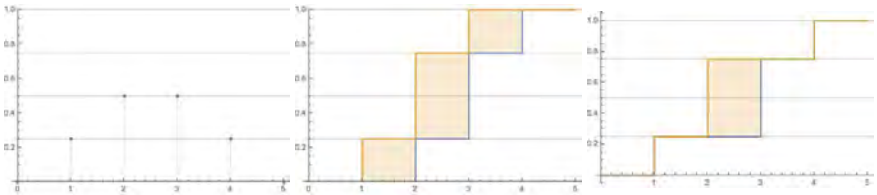
*where, for each approximation, the following must always hold:*

$$\forall z. 0 \leq \overline{F}_{\texttt{p}}(z) \leq 1 \qquad \forall z. 0 \leq \underline{F}_{\texttt{p}}(z) \leq 1$$

When we can deduce that a program may return one of two values but not which value, the cumulative probability can be used for a more precise description. Consider the following program, `test`, which contains an unanalysable test, indicated by '`(* unanalysable *)`'.

```
test(x) =
   if x = 1 then 1
   else (if x = 4 then 4
         else (if (* unanalysable *) then 2
               else 3))
```

Let the input probability distribution be $P(x) = 1/4 \cdot C(1 \leq x \leq 4)$. Then, the tightest possible over-approximating output probability distribution for `test` is $\overline{P}_{\texttt{test}}$, as depicted in Figure 3.1(a). The functions $F^{\uparrow}_{\texttt{test}}$ and $F^{\downarrow}_{\texttt{test}}$ are defined solely by $\overline{P}_{\texttt{test}}$ (see Definition 3.8) and are themselves over-approximations and under-approximations of $F_{\texttt{p}}$, respectively. The $F^{\uparrow}_{\texttt{test}}$ and $F^{\downarrow}_{\texttt{test}}$ are depicted in Figure 3.1(b), where the yellow area can be interpreted as their imprecision. However,



(a) (b) (c)

Fig. 3.1: (a) depicts the output probability distribution of `test`. (b) depicts the functions $F^{\uparrow}_{\texttt{p}}$ (yellow), the over-approximation of $F_{\texttt{p}}$, and $F^{\downarrow}_{\texttt{p}}$ (blue), the under-approximation of $F_{\texttt{p}}$. (c) depicts the tightest possible over- and under-approximation of the cumulative distribution `test` with the available analysable information.

$F^{\uparrow}_{\texttt{test}}$ and $F^{\downarrow}_{\texttt{test}}$ are not as tight as when the cumulative distributions can be derived directly. In Figure 3.1(c), we have depicted the tightest possible cumulative output distributions for `test`. These boundaries exploit the fact that the input $1$ may only relate to output $1$ and that input $4$ may only relate to output $4$; thus, the probability of output $1$ is $25\%$, and the probability of $4$ is $25\%$.

**Approximations for cumulative probability functions.**

When approximating cumulative probability functions, the techniques are different from probability mass functions. Instead, one may use copulas [17] to over- and under-approximate dependencies between subexpressions. Copulas are based on the theory of comonotonicity [38] for distributions that may depend on a common (possibly unknown) random variable.

## 3.8 Related Work

Probabilistic analysis is related to the analysis of probabilistic programs. Probabilistic analysis is the analysis of programs with a normal semantics where the input variables are interpreted over probability distributions. The analysis of probabilistic programs analyses programs with probabilistic semantics where the values of the input variables are unknown (e.g., flow analysis [108]).

In probabilistic analysis, it is important to determine how variables depend on each other; however, already in 1976, Denning proposed a flow analysis for revealing whether variables depend on each other [36]. This was presented in the field of secure flow analysis. Denning introduced a lattice-based analysis where she, given the name of a variable, which should be kept secret, deducted which other variables should be kept secret to avoid leaking information. In 1996, Denning's method was refined by Volpano *et al*. into a type system, and for the first time, it was proven sound [139].

Reasoning about probabilistic semantics is a closely related area to probabilistic analysis, as they both work with nested probabilistic influence. The probabilistic analysis works on standard semantics and analyses it using input probability distributions, where a probabilistic semantics allows for random assignments and probabilistic choices [83] and is normally analysed using an expanded classical analysis or verification method [33].

Probabilistic model checking is an automated technique for formally verifying quantitative properties for systems with probabilistic behaviours. It is mainly focused on Markov decision processes, which can model both stochastic and nondeterministic behaviours [52, 84] This differs from probabilistic analysis, as it assumes the Markov property.

In 2000, Monniaux applied abstract interpretation to programs with probabilistic semantics and obtained safe bounds for worst-case analysis [95]. Pierro *et al*. introduced a linear mapping structure, a Moore-Penrose pseudo-inverse, instead of a Galois connection. They used the linear structures to compare the 'closeness' of approximations as an expression using the average approximation error. Pierro *et al*. further explored using probabilistic abstract interpretation to conduct the average-case analysis [40]. In 2012, Cousot and Monerau gave a general probabilistic abstraction framework [33] and stated, in section 5.3, that Pierro *et al*.'s method and many other abstraction methods can be expressed in this new framework.

When analysing probabilities, the main challenge is to maintain the dependencies throughout the program. Schellekens defines this as *Randomness preservation* [124] (or random bag preservation), which in his (and Gao's [57]) case enables the tracking of certain data structures and their distributions. They use special data structures since they find these suitable to derive the average number of basic operations. In another approach [109, 141], tests in programs have been assumed to be independent of previous history, also known as the Markov property (the probability of being true is fixed). As Wegbreit remarked, this is true only for certain programs (e.g., linear search for repeating lists), and for other programs, this is not the case (linear search for non-repeating lists). The Markov property is the foundation of Markov decision processes, which are used in probabilistic model checking [52]. Cousot et

al. presented a probabilistic abstraction framework wherein they divided the program semantics into probabilistic and (non-)deterministic behaviours. They proposed the handling of tests when it is possible to assume the Markov property, and they handle loops by using a probability function describing the probability of entering the loop in the $i$th iteration. Monniaux proposed another approach for abstracting probabilistic semantics [95]; he first lifts a normal semantics to a probabilistic semantics whereby random generators are allowed and then uses an abstraction to achieve a closed form. Monniaux's semantic approach uses a backward probabilistic semantics operating on measurable functions. This is closely related to the forward probabilistic semantics proposed earlier by Kozen [83].

An alternative approach to probabilistic analysis is based on the symbolic execution of programs with symbolic values [58]. Such techniques can also be used on programs with infinitely many execution paths by limiting the analysis to a finite sets of paths at the expense of tightness of probability intervals [120].

## 3.9 Conclusion

The probabilistic analysis of programs has seen renewed interest for analysing programs with respect to energy consumption. Numerous embedded systems and mobile applications are limited by restricted battery life. In this paper, we present a technique for extracting a probability distribution for programs from symbolic distributions of the input. This technique is a transformation-based method, whereby we analyse a first-order language with a simple tail recursion. From the original program, we generate an equivalent probability distribution program and transform this program into a closed form. We present the essential transformation rules for unfolding calls to the original program and removing infinite sums. The transformed program may then be analysed and approximated using program analysis and transformation techniques known from automatic complexity analysis. The core elements of the analysis have been implemented in a prototype system with the aim of using the analysis to improve the energy efficiencies of systems. The central challenges of approximating in a probabilistic setting are discussed, and we describe some advantages of using cumulative distributions along with copulas to achieve a tighter approximation.

**Acknowledgements.** This work has benefited from numerous discussions with Pedro López-García, Alejandro Serrano Mena and other colleagues in Madrid, Bristol and Roskilde.

## 3.10 Afterword

In the article, we suggest several interesting concepts and challenges for future work: analysing the *expected output*, which is the weighted average of the output; *externalize resource usage* and analysing resource-instrumented programs; the handling of input with a probability that yields a *non-terminating* computation; approximating

the relations between variables using *copulas*, which are functions that describe a joint distribution based on two independent cumulative distributions, *non-primitive types*, such as lists and *P-boxes*, that are upper and lower bounds for cumulative distributions. There are two issues that we have not followed up on: expected outputs and copulas. The first challenge that we addressed was externalizing resource usage and analysing resource-instrumented programs; in Chapter 4, we develop a discrete probabilistic resource analysis. The lists are not addressed in the discrete case, but they are captured in the general theory of Chapter 5. P-boxes will not be used directly, but they raised the question of how relations and probabilities were connected, which ultimately lead to the findings presented in Chapter 5. In that chapter, we also discuss and demonstrate the techniques on a non-terminating program (Section 5.5.3). In Chapter 8, we return to discuss the concept of copulas peripherally.

# 4

# Discrete probabilistic resource analysis

In this chapter, we apply discrete output probability analysis (Chapter 3) to resource-instrumented programs and obtain a discrete probabilistic resource analysis. This analysis can be applied to terminating and deterministic programs, and it yields the upper probability bounds for their resource usages. Given a program, the analysis instruments it with a discrete resource model, here a step counter, and slices the instrumented version to output only the resource usage. Then, it applies the probabilistic output analysis to obtain upper probability bounds on the resource usage.

*Notation and Terminology.* The notation and terminology are like those of the previous chapter, that is, in the following article, we use "an over-approximation" when referring to an upper bound of a function and "an under-approximation" when referring to a lower bound of a function; see for instance Definition 4.4. Note that Section 4.2 contains selected definitions from Section 3.2.

*Foreword.* The remainder of this chapter – except for the afterword (Section 4.9) – has been published with minor corrections in article [78] M. H. Kirkeby, M. Rosendahl. Probabilistic resource analysis by program transformation. M. van Eekelen, U. Dal Lago, redaktorzy, *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, strony 60–80, Cham, 2016. Springer International Publishing.

**Abstract.** The aim of a probabilistic resource analysis is to derive a probability distribution of possible resource usage for a program from a probability distribution of its input. We present an automated multi-phase rewriting-based method to analyse programs written in a subset of C. The method generates a probability distribution of the resource usage as a possibly uncomputable expression and then transforms it into a closed-form expression using over-approximations. We present the technique, outline the implementation and show results from experiments with the system.

## 4.1 Introduction

The main contribution of this paper is to present a technique for probabilistic resource analysis whereby the analysis is seen as a program-to-program translation. This means that the transformation to closed form is a source code program transformation problem and not specific to the analysis. Any necessary approximations in the analysis are performed at the source-code level. The technique also makes it possible to balance the precision of the analysis against the brevity of the result.

Many optimizations for increased energy efficiency require probabilistic and average case analysis as part of the transformations. Wierman et al. state that *"global energy consumption is affected by the average case, rather than the worst case"* [144]. In addition, in scheduling, *"an accurate measurement of a task's average-case execution time can assist in the calculation of more appropriate deadlines"* [62]. For a subset of programs, a precise average-case execution time can be found using static analysis [51, 57, 124]. Applications of such analysis may be in improving the scheduling of operations or in temperature management. Because the analysis returns a distribution, it can be used to calculate the probability of energy consumptions above a certain limit, thereby indicating the risk of over-heating.

The central idea in this paper is to use probabilistic output analysis in combination with a preprocessing phase that instruments programs with resource usage. We translate programs into an intermediate language program that computes the probability distribution of resource usage. This program is then analysed, transformed, and approximated with the objective of obtaining a closed-form expression. This is an alternative to deriving cost relations directly from the program [35] or expressing costs as abstract values in a semantics for the language.

As with automatic complexity analysis, the aim of probabilistic resource analysis is to express the result as a parameterized expression. The time complexity of a program should be expressed as a closed-form expression in the input size, and for probabilistic resource analysis, the aim is to express the probability of resource usage of the program parameterized by input size or range. If input values are not independent, we can specify a joint distribution for the values. Values do not have to be restricted to a finite range; however, for infinite ranges, the distribution would converge to zero in the limit.

The current work extends our previous work on probabilistic analysis [118] in three ways. We show how to use a preprocessing phase to instrument programs with resource usage such that the resource analysis can be expressed as an analysis of possible outputs of a program. The resource analysis can handle an extended class of programs with structured data as long as the program flow does not depend on the probabilistic data in composite data structures. Finally, we present an implementation of the analysis in the Ciao language [21], which uses algebraic reductions available in the Mathematica system [146].

The focus in this paper is on using fairly simple local resource measures where we count core operations on data. Since the instrumentation is done at the source-code level, we can use flow information so that the local costs can depend on actual data on operations and which operations are executed before and after. This is not

normally relevant for time complexity but does play an important role in energy consumption analysis [76, 135].

## 4.2 Probability distributions in static analysis

In our approach to probabilistic analysis, the result of an analysis is an approximation of a probability distribution. We will here present the concepts and notation that we will use in the remainder of the paper. A probability distribution is also often referred to as the *probability mass function* in the discrete case; in the continuous case, it is a *probability density function*. We will use an upper case $P$ to denote a probability distribution.

**Definition 4.1 (probability distribution).** *For a countable set $X$, a probability distribution over $X$ is a mapping $P_X \colon X \to [0,1]$, where*

$$\sum_{x \in X} P_X(x) = 1$$

We define the output probability distribution for a program $\mathtt{p}$ in a forward manner. This is the *weight* or sum of all probabilities of input values, where the program returns the desired value $z$ as output.

**Definition 4.2 (output probability).** *Given a program, $\mathtt{p} \colon X \to Z$, and a probability distribution over the input, $P_X$, the probability distribution over output, $P_{\mathtt{p}} \colon Z \to [0,1]$, is defined as*

$$P_{\mathtt{p}}(z) = \sum_{x \in X \wedge \mathtt{p}(x) = z} P_X(x)$$

Note that Kozen also uses a similar forward definition [83], whereas Monniaux constructs the inverse mapping from output to input for each program statement and expresses the relationship in a backwards style [95].

**Lemma 4.3.** *The output probability distribution, $P_{\mathtt{p}} \colon Z \to [0,1]$, satisfies*

$$0 \leq \sum_{z \in Z} P_{\mathtt{p}}(z) \leq 1$$

The program may not terminate for all input, and this means that the sum may be less than one. If we expand the domain $Z$ with an element to denote non-termination, $Z_\perp$, the total sum of the output distribution $P_{\mathtt{p}}(z)$ would be 1.

In our static analysis, we will use approximations to obtain safe and simplified results.

**Definition 4.4 (over- and under-approximation).** *Let $P_{\mathtt{p}} \colon Z \to [0,1]$ be a distribution, and let $\overline{P}_{\mathtt{p}} \colon Z \to [0,1]$ be a function; then, $\overline{P}_{\mathtt{p}}$ is an* over-approximation of *$P_{\mathtt{p}}$ if $P_{\mathtt{p}}(z) \leq \overline{P}_{\mathtt{p}}(z) \leq 1$.*

The aim of the probabilistic resource analysis is to derive as tight an approximation $\overline{P}_{\mathrm{p}}$ as possible.

The over-approximation of the probability distribution can be used to derive lower and upper bounds of the expected value and will thus approximate the expected value as an interval [118].

## 4.3 Architecture of the transformation system

The system contains five main phases. The input to the system is a program in a small subset of C with annotations of which part we want to analyze. It could be the whole program, but it could also be a specific subroutine that is called repeatedly with varying arguments according to some input distribution.

The first phase will instrument the program with resource-measuring operations. The instrumented program will perform the same operations as the original program in addition to recording and printing resource usage information. This program can still be compiled and run, and it will also produce the same results as the original program.

The second phase translates the program into an intermediate language for further analysis. We use a small first-order functional language for the analysis process. The translation has two core elements. We slice [142] the program with respect to the resource-measuring operations and transform loops into a simple form of tail recursion in the intermediate language. The transformed program can still be executed and will produce the same resource usage information as the instrumented program. Since the instrumentation is performed before the translation into the intermediate language, any interpretation overhead or speed-up due to slicing does not influence the result [116].

In the third phase, we construct a probability output program that computes the probability output function. In this case, it is a probability distribution of possible resource usages of the original program. This program can also run but will often be extremely inefficient since it will merge information for all possible inputs to the original program.

The fourth phase transforms the probability program into a large expression without further function calls. Recursive calls are removed using summations, and the transformed program computes the same result as the program did before this phase.

In the final phase, the probability function is transformed into closed form using symbolic summation and over-approximation. In this phase, we use the Mathematica system [146]. The final probability program computes the same result or an over-approximation of the function produced in the fourth phase.

## 4.4 Instrumenting programs for resource analysis

The input to the analysis is a program in a subset of C. In the next section, we define the intermediate language for further analysis, and it is the restrictions on the

intermediate language that limit the source programs that we can analyze with our system. The source program may contain integer variable and arrays, typical loop constructs and non-recursive function calls. The program should be annotated with specifications on which part of the program to analyse. The following is an example of such a program.

```
// ToAnalyse: multa(_,_,_,N)
void multa(int a1[MX],int a2[MX],int a3[MX],int n){
  int i1,i2,i3,d;
  for(i1 = 0; i1 < n; i1++) {
    for(i2 = 0; i2 < n; i2++) {
      d = 0;
      for(i3 = 0; i3 < n; i3++) {
         d = d + a1[i1*n+i3]*a2[i3*n+i2];
      }
      a3[i1*n+i2] = d;
    }
  }
}
```

This example program describes a matrix multiplication for which we would like to analyse the probability distribution of the number of steps when parameterized with the size (`N`) of the matrices.

### 4.4.1 Instrumentation

The program is then instrumented with resource usage information and translated into an intermediate language for further analysis. The instrumented program is also a valid program in the source language and can be executed to obtain the same results as the original program. It will, however, also collect resource usage information.

In our example, we instrument the program with step counting information whereby we count the number of assignment statements being executed. This is done by inserting a variable into the program and incrementing it once for each assignment statement.

```
int multa(int a1[MX],int a2[MX],int a3[MX],int n){
  int i1,i2,i3,d;
  int step; step=0;
  for(i1 = 0; i1 < n; i1++) {
    for(i2 = 0; i2 < n; i2++) {
      d = 0; step++;
      for(i3 = 0; i3 < n; i3++) {
         d = d + a1[i1*n+i3]*a2[i3*n+i2]; step++;
      }
      a3[i1*n+i2] = d; step++;
    }
  }
  return step;
}
```

The outer loop does not update the step counter, whereas the first inner loop updates it twice per iteration, and the innermost loop updates it once per loop iteration.

### 4.4.2 Slicing

The second phase will slice, *e.g.* [142], the program with respect to resource usage and translate the program into the intermediate language of first-order functions that we will use in the subsequent stages. Loops in the program are translated into a simple recursive pattern.

```
for3(i3, step, n) =
  if(i3 = n) then step else for3(i3 + 1,step+1,n)

for2(i2, step, n) =
  if(i2 = n) then step else for2(i2 + 1,for3(0,step+2,n),n)

for1(i1, step, n) =
  if(i1 = n) then step else for1(i1 + 1,for2(0,step,n),n)

tmulta(n)= for1(0,step,n)
```

Each function in the recursive program corresponds to a for loop with their related step updates. The step counter is given as an input argument to the next function in a continuation-passing style.

### 4.4.3 Intermediate language

An intermediate program, $prg_f$, consists of a series of integer functions, $\mathtt{f_i}\colon Int^* \to Int$, as described by the abstract syntax in Figure 4.1. In the examples, we relax the restrictions on function and parameter names.

$$
\begin{array}{lll}
a & ::= & \mathtt{x} \mid n \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1 \,\mathtt{div}\, a_2 \\
b & ::= & a_1{=}a_2 \mid a_1{<}a_2 \mid a_1{=}{<}a_2 \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{not}(b) \\
e & ::= & a \mid \mathtt{f}(e_1, \ldots, e_n) \mid \mathtt{if}\ b\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\
f & ::= & \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = e \\
prg_f & ::= & f \mid ff
\end{array}
$$

Fig. 4.1: The abstract syntax for describing functions of the intermediate programs, where $a$ is an arithmetic integer expression, $b$ is a boolean expression, and $e$ is an statement expression. $n$ is a numeral, an integer, and $\mathtt{x}_i$ are variable names.

**Definition 4.5.** *A program is* well-formed *if it follows the abstract syntax and contains a finite number of function definitions, where each is of one of the following forms and can internally be enumerated with a natural number such that*

$\mathtt{f}_i(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{if}\ b\ \mathtt{then}\ e_0\ \mathtt{else}\ \mathtt{f}_i(e_1, \ldots, e_n)$
*where* $\mathtt{f}_i$ *is simple,* $e_0$ *only contains calls to functions* $\mathtt{f}_j$ *where* $j < i$.

$\mathtt{f}_i(\mathtt{x}_1, \ldots, \mathtt{x}_n) = e$
*where* $e$ *only contain calls to functions* $\mathtt{f}_j$ *where* $j < i$.

The enumeration prevents mutual recursion and ensures that non-recursive calls cannot create an infinite call chain.

## 4.5 Probabilistic output analysis

The analysis is applied to the intermediate program and an input probability program in the intermediate language. The output is a new program that can be described by a subset of the intermediate language; this will be clarified later in the definition of pure and closed-form programs. The analysis consists of three phases:

Create, where the probability program describing the output distribution is created as a possibly uncomputable expression.
Separate, where we remove all calls from the probability program.
Simplify, where we transform the program into closed form using safe over-approximations when necessary.

The analysis is constructed as three sets of transformation rules, one for each of the three phases. All transformations are syntax directed, and one strategy is to apply them in a depth-first manner. The program output analysis is implemented in Ciao and uses Mathematica [146] as an external solver in the third phase to reduce expressions.

In the following, we use $\mathcal{V}ar(e)$ to represent the set of variables occurring in expression $e$, and we use $\mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n) \stackrel{\text{def}}{=} e$ to represent the function `f` defined in the input program. Some side conditions are explained in an informal manner, as in "$\mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n) \stackrel{\text{def}}{=} e$, where $e$ is non-recursive".

$$\text{name}\ \frac{\text{precondition}_1 \quad \ldots \quad \text{precondition}_n}{\text{original term} \rightarrow \text{rewritten term}}$$

The preconditions are evaluated from left to right, and if all succeed, we can use the transformation.When substituting a variable `x` in an expression $e$, we denote it $[\mathtt{x}/e]$.

In the following, we will begin by extending the intermediate language presented in Figure 4.1 such that it can express probabilities, and then, we describe the transformation rules for each phase.

### 4.5.1 The intermediate language

The intermediate language is, as previously mentioned, a first-order functional language. A probability program can be evaluated at any stage through the transformation process.

We extend the abstract syntax given in Figure 4.1 such that it can easily describe probability distributions. We introduce probability functions, $\texttt{P}\colon \mathit{Int}^* \to \mathit{Real}$, which follows the expanded syntax given in Figure 4.2. The dots indicate the syntax described in Figure 4.1. Again, ⟨aexp⟩ and ⟨exp⟩ are of the integer type, ⟨bexp⟩ is boolean, and the new ⟨qexp⟩ is a real. In ⟨qexp⟩, the method `i2r` type casts an integer expression to a real. We reuse the arithmetic operator symbols for the reals and let the types of their arguments distinguish the integer operators and the real operators. We introduce `c`, `sum`, `prod` and `argDev` functions. `c` evaluates to either 1, if its boolean expression evaluates to true, or 0 when it evaluates to false. Evaluating `sum` instantiates the variable with all possible values and sums all the results of the evaluation with ⟨qexp⟩. `prod` instantiates its variable with all values for which the first ⟨qexp⟩ evaluates to 1, and then, it multiplies all the results from evaluating the second ⟨qexp⟩. The last expression introduced is `argDev`, which describes the development of the variable $\texttt{x}_i$ as a function of the number of updates, $\texttt{x}_j$. The expression ⟨exp⟩ computes the development of $\texttt{x}_i$ for one increment of $\texttt{x}_j$ (e.g., the argument $\texttt{x}_i$ in a function $\texttt{f}(\texttt{x}_i)$ with a recursive call $\texttt{f}(\texttt{x}_i - 2)$ has an argument development $\texttt{argDev}\,(\texttt{x}_i, \texttt{x}_i{-}2, \texttt{x}_j)$). A program that computes a probability

$$
\begin{array}{lcl}
a & ::= & \ldots \mid \texttt{min}(a_1, a_2) \mid \texttt{max}(a_1, a_2) \\
b & ::= & \ldots \mid a = e \\
e & ::= & \ldots \mid \texttt{argDev}(\texttt{x}_1, e, \texttt{x}_2) \\
q & ::= & \texttt{i2r}(q) \mid \texttt{c}(b) \mid q_1 + q_2 \mid q_1 - q_2 \mid q_1 * q_2 \mid q_1 \,/\, q_2 \\
 & & \texttt{sum}(\texttt{x}, q) \mid \texttt{prod}(\texttt{x}, q_1, q_2) \mid \texttt{P}(a_1, \ldots, a_n) \\
p & ::= & \texttt{P}(\texttt{x}_1, \ldots, \texttt{x}_n) = q \\
prg_f & ::= & f \mid ff \\
prg_p & ::= & p \mid pp \\
prg & ::= & prg_p\, prg_f
\end{array}
$$

Fig. 4.2: The expanded abstract syntax describing probability programs. We reuse the arithmetic operator symbols; letting the type of arguments distinguish the operators.

distribution is referred to as a probability program.

**Definition 4.6.** *A probability program that has no if-expressions and no function calls is* pure*, and a pure probability program without any* `sum` *or* `prod` *is in* closed form*.*

A program is *pure* after it is transformed in the separation phase and is pure and in *closed form* after the simplification phase.

### 4.5.2 The create phase

This phase has only one rule, which creates a program that computes a probability distribution from the intermediate program and input distributions.

$$\text{create}\ \frac{\mathtt{f}(\mathtt{u}_1,...,\mathtt{u}_n)\ = e_f \qquad \mathtt{P}(\mathtt{v}_1,...,\mathtt{v}_n) = e_p}{P_{\mathtt{f}}(\mathtt{z})\ =\ \mathtt{sum}(\mathtt{x}_1\,;...\mathtt{sum}(\mathtt{x}_n\,;\mathtt{c}(\mathtt{z}{=}\,\mathtt{f}(\mathtt{x}_1,...,\mathtt{x}_n)\,)\ \ast\ \mathtt{P}(\mathtt{x}_1,...,\mathtt{x}_n)))}$$

We use the create rule to make a new probability function describing the probability distribution for the integer function in which we are interested.

### 4.5.3 The separate phase

In this phase, function calls are removed by repeatedly exposing calls and replacing them. Non-recursive function calls are unfolded using their definitions. Function calls can occur inside if-expressions or as nested calls; these are extracted and handled one at a time.

$$\text{f-simple}\ \frac{\mathtt{f}(\mathtt{y}_1,...,\mathtt{y}_n)\ = e \quad \text{, where } e \text{ is non-recursive} \quad \mathtt{x}_1,...,\mathtt{x}_n \in \mathcal{V}ar}{\mathtt{c}(\mathtt{z}{=}\,\mathtt{f}(\mathtt{x}_1,...,\mathtt{x}_n)) \rightarrow \mathtt{c}(\mathtt{z}{=}\,e[\mathtt{y}_1/\mathtt{x}_1,...,\mathtt{y}_n/\mathtt{x}_n])}$$

$$\text{rem-P}\ \frac{\mathtt{P}(\mathtt{x}_1,...,\mathtt{x}_n) = e}{\mathtt{P}(e_1,...,e_n) \rightarrow e[\mathtt{x}_1/e_1,...,\mathtt{x}_n/e_n]}$$

$$\text{rem-if}\ \frac{}{\mathtt{c}(\mathtt{z}{=}\,\mathtt{if}\ b\ \mathtt{then}\ e_0\ \mathtt{else}\ e_1) \rightarrow (\mathtt{c}(b)\ \ast\ \mathtt{c}(\mathtt{z}{=}e_0) + \mathtt{c}(\mathtt{not}(b))\ \ast\ \mathtt{c}(\mathtt{z}{=}e_1))}$$

$$\text{no-nest(f)}\ \frac{e_1,...,e_n \notin \mathcal{V}ar}{\begin{array}{l}\mathtt{c}(\mathtt{z}{=}\,\mathtt{f}(e_1,...,e_n)\,) \rightarrow \\ \mathtt{sum}(\mathtt{u}_1\,;...\mathtt{sum}(\mathtt{u}_n\,;\mathtt{c}(\mathtt{z}{=}\,\mathtt{f}(\mathtt{u}_1,...,\mathtt{u}_n)\,)\ \ast\ \mathtt{c}(\mathtt{u}_1{=}\,e_1)\ \ast\ ...\ \ast\ \mathtt{c}(\mathtt{u}_n{=}\,e_n)\,))\end{array}}$$

We replace calls to recursive functions by a summation over the number of recursions using argument development constructors to describe the value of each argument as a function of the index of the summation. This way of defining argument development has similarities with size change functions derived using recurrence equations. Argument development functions do not depend on the base case, in contrast to size-change functions [149]. The summation also contains a product that ensures that the condition evaluates to false for argument values less than the current value of the index of summation. When the expression in a product contains only c-constructors, then the product is evaluated to 1 if either the range is empty or the expression is evaluated to true for the full range. The following rewrite rules are all that is needed for transforming probability programs into pure probability programs.

$$\text{f-rec}\ \frac{\begin{array}{c}\mathtt{f}(\mathtt{y}_1,...,\mathtt{y}_n) = \mathtt{if}\ b\ \mathtt{then}\ e_0\ \mathtt{else}\ \mathtt{f}(e_1,...,e_n) \qquad \mathtt{x}_1,...\mathtt{x}_n \in \mathcal{V}ar \\ \sigma_{y/i} = [\mathtt{y}_1/\mathtt{i}_1,...,\mathtt{y}_n/\mathtt{i}_n]\sigma_{y/x} = [\mathtt{y}_1/\mathtt{x}_1,...,\mathtt{y}_n/\mathtt{x}_n]\sigma_{y/j} = [\mathtt{y}_1/\mathtt{j}_1,...,\mathtt{y}_n/\mathtt{j}_n]\end{array}}{\begin{array}{l}\mathtt{c}(\mathtt{z}{=}\,\mathtt{f}(\mathtt{x}_1,...,\mathtt{x}_n)) \rightarrow \\ \mathtt{sum}(\mathtt{i}\,;\mathtt{c}(0{=}{<}\mathtt{i})\ \ast \\ \quad \mathtt{sum}(\mathtt{i}_1\,;...\mathtt{sum}(\mathtt{i}_n\,;\mathtt{c}(\sigma_{y/i}(b))\ \ast\ \mathtt{c}(\mathtt{i}_1{=}\,\mathtt{argDev}(\mathtt{x}_1,\sigma_{y/x}(e_1),\mathtt{i}))\ \ast \\ \qquad \mathtt{c}(\mathtt{z}{=}\,\sigma_{y/i}(e_0))\ \ast\ ...\ \ast\ \mathtt{c}(\mathtt{i}_n{=}\,\mathtt{argDev}(\mathtt{x}_n,\sigma_{y/x}(e_n),\mathtt{i}))\ )...)\ \ast \\ \quad \mathtt{prod}(\mathtt{j}\,;\mathtt{c}(0{=}{<}\mathtt{j})\ \ast\ \mathtt{c}(\mathtt{j}{=}{<}\mathtt{i}-1)\,; \\ \quad \mathtt{sum}(\mathtt{j}_1\,;...\mathtt{sum}(\mathtt{j}_n\,;\mathtt{c}(\mathtt{not}(\sigma_{y/j}(b)))\ \ast \\ \qquad \mathtt{c}(\mathtt{j}_1{=}\,\mathtt{argDev}(\mathtt{x}_1,\sigma_{y/x}(e_1),\mathtt{j}))\ \ast\ ...\ \ast\ \mathtt{c}(\mathtt{j}_n{=}\,\mathtt{argDev}(\mathtt{x}_n,\sigma_{y/x}(e_n),\mathtt{j})) \\ \quad )...)\,))\end{array}}$$

The argument development expression may contain function calls as well, and these are extracted equivalently to nested functions.

no-nest(argDev)
$$\texttt{c}(\texttt{z}{=}\,\texttt{argDev}(\texttt{x},\texttt{f}(e_1,...,e_n)\,,\texttt{i})) \rightarrow$$
$$\texttt{sum}(\texttt{u}\,;\texttt{c}(\texttt{z}{=}\,\texttt{argDev}(\texttt{x},\texttt{f}(e_1,...,e_n)\,,\texttt{i}))\star\texttt{c}(\texttt{u}{=}\,\texttt{f}(e_1,...,e_n))\,)$$

Once these rules can no longer be applied, the probability program has been transformed to pure form.

### 4.5.4 The simplification phase

We have presented the rules for obtaining a pure probability program, and in this section, we outline the rules used to reach closed form. A pure probability function consists of a series of nested summations multiplied with an expression (e.g., input probability). The rules are applied in no particular order, and the phase ends when no more rules can be applied. In this phase, we employ Mathematica [146]. A call to Mathematica is denoted $\texttt{mm:Function}(Arg) = Answer$, where `Function` denotes the actual function called in Mathematica (e.g., `mm:Expand` calls Mathematica's `Expand` function). The translation between the intermediate language and Mathematica's representation will not be discussed further here. implicitly in the call.

The rules can be grouped by their functionality: preparing expressions, removal of summations and removal of products. The rules for removal of products are currently the only rules containing over-approximations.

**Preparation.** Preparing expressions for the removal of either summations or products involves moving expressions that do not depend on the index of summation outside the summation, dividing summations of additions into simpler ones, reducing expressions, dividing summations in ranges, and removing argument development constructors. Note that div-sum(x$\leq$) has an equivalent rule for upper bounds.

$$\text{move-c}\frac{\mathtt{x} \notin \mathcal{V}ar(e_1)}{\mathtt{sum}(\mathtt{x}\,;\, e_1 \star e_2) \to e_1 \star \mathtt{sum}(\mathtt{x}\,;\, e_2)}$$

$$\text{div-sum(+)}\frac{\mathtt{x} \in \mathcal{V}ar(e_1) \qquad x \in \mathcal{V}ar(e_2)}{\mathtt{sum}(\mathtt{x}\,;\, e_1 + e_2) \to \mathtt{sum}(\mathtt{x}\,;\, e_1) + \mathtt{sum}(\mathtt{x}\,;\, e_2)}$$

$$\text{div-sum(x}\leq\text{)}\frac{x \notin \mathcal{V}ar(e_1, e_2) \qquad x \in \mathcal{V}ar(e_2)}{\begin{array}{l}\mathtt{sum}(\mathtt{x}\,;\, \mathtt{c}(\mathtt{x}{=}{<}e_1) \star \mathtt{c}(\mathtt{x}{=}{<}e_2) \star e_3) \to \\ \mathtt{c}(e_1{=}{<}e_2) \star \mathtt{sum}(\mathtt{x}\,;\, \mathtt{c}(\mathtt{x}{=}{<}e_1) \star e_3) + \\ \mathtt{c}(e_2{=}{<}e_1 - 1) \star \mathtt{sum}(\mathtt{x}\,;\, \mathtt{c}(\mathtt{x}{=}{<}e_2) \star e_3)\end{array}}$$

$$\text{rem(argDev)}\frac{\mathtt{c}(y = e) \to \mathtt{c}(y{=}\,\mathtt{x} + a)}{\mathtt{c}(\mathtt{z}{=}\,\mathtt{argDev}(\mathtt{x}, e, \mathtt{i})) \to \mathtt{c}(\mathtt{z}{=}\,\mathtt{x} + a \star \mathtt{i})}\ \mathcal{V}ar(a) = \emptyset$$

$$\text{reduceAexp}\frac{\mathtt{mm{:}Reduce}(e_1) \equiv e_2}{\mathtt{c}(e_1) \to \mathtt{c}(e_2)}$$

$$\text{reduce(=)}\frac{}{\mathtt{c}(true) \to \mathtt{i2r}(1)}$$

**Summations.** Removal of summations can be done in two ways. Either the index of the summation can only be one value or it can be a limited range of values; depending on the case, different transformations are used. In the first case, there exists an equation containing the variable index of the innermost summation. The equation is solved for the variable, and the remaining variable occurrences are replaced by the new value.

$$\text{rem-sum(=)}\frac{\mathtt{mm{:}Solve}(e_1{=}e_2, \mathtt{x}) \equiv e_3}{\mathtt{sum}(\mathtt{x}\,;\, \mathtt{c}(e_1{=}\,e_2) \star e) \to e[\mathtt{x}/e_3]}$$

Removing a summation by its range involves using standard mathematical formulas for rewriting series. The last part of the following rule uses $\sum_{k=1}^{n} k^2 = n(n+1)(2n+1)/6$. We only present transformations up to quadratic series, and our pragmatic implementation contains rules for transforming series of power of degree up to 10. A more general rewrite rule for series of power of degree up $p$ could be implemented, but is more complicated, as it includes Bernoulli numbers and binomial coefficients. The precondition uses Mathematica's `Expand` function [146] to transform the expression into the correct pattern.

$$\text{rem-sum}(\leq)\frac{\mathtt{x} \notin \mathcal{V}ar(e_1,...,e_6)\ \ \mathtt{mm{:}Expand}(e_3) \equiv e_4 + e_5 \star \mathtt{x} + e_6 \star \mathtt{x} \star \mathtt{x}}{\begin{array}{l}\mathtt{sum}(\mathtt{x}\,;\mathtt{c}(e_1{=}{<}\mathtt{x}) \star \mathtt{c}(\mathtt{x}{=}{<}e_2) \star \mathtt{i2r}(e_3)) \rightarrow \\ \quad \mathtt{c}(e_1{=}{<}e_2) \star \big(\mathtt{i2r}(e_4) \star \mathtt{i2r}(e_2 - e_1 + 1) + \\ \quad \mathtt{i2r}(e_5) \star \mathtt{i2r}(e_2 \star (e_2{+}1))/\,\mathtt{i2r}(2) - \\ \quad \mathtt{i2r}(e_5) \star \mathtt{i2r}(e_2 \star (e_2 - 1))/\,\mathtt{i2r}(2) + \\ \quad \mathtt{i2r}(e_6) \star \mathtt{i2r}(e_2 \star (e_2{+}1) \star (2 \star e_2 + 1))/\,\mathtt{i2r}(6) - \\ \quad \mathtt{i2r}(e_6) \star \mathtt{i2r}(e_2 \star (e_2{-}1) \star (2 \star e_2 - 1))/\,\mathtt{i2r}(6)\big)\end{array}}$$

**Product.** The removal of Product involves a safe approximation. The implementation of POA contains two different over-approximations, and in many cases, the probability program can be transformed into closed form in a precise manner. In the following paragraph, we describe when the transformation preserves the accuracy of the transformed term.

The probability function can always be over-approximated to 1. The rule f-rec is an exact rule and introduces a product-expression that may not be possible to rewrite into closed form. We only introduce the product-expression with c-expressions in its body, and therefore, it may at most produce 1 and at least produce 0. The following rule over-approximates a product-expression to 1.

$$\text{rem-prod-one}\frac{\mathtt{x} \notin \mathcal{V}ar(e_1,e_2) \qquad \mathtt{x} \in \mathcal{V}ar(q)}{\mathtt{prod}(\mathtt{x}\,;\,\mathtt{c}(e_1{=}{<}\mathtt{x}) \star \mathtt{c}(\mathtt{x}{=}{<}e_2)\,;\,q) \rightarrow 1}$$

For the summation describing recursive calls, this transformation is exact when the condition $b$ evaluates to true for exactly one value (e.g., it is an equation).

A broader class of recursive programs (than those having an equation in the condition) is that where the c-expression is monotone in $x$, meaning that there exists a $k$ for which $c(e_3) = 1$ for $x \leq k$ and $c(e_3) = 0$ for $x > k$. This case covers many for-loops. In this case, we can accurately replace the prod-expression with two c-expressions, one checking the lower range limit and one checking the upper range limit. The empty product (the lower limit is larger than the upper) is 1.

$$\text{rem-prod-mon}\frac{\mathtt{x} \notin \mathcal{V}ar(e_1,e_2) \qquad \mathtt{x} \in \mathcal{V}ar(e_3) \qquad q \text{ is monotone in } \mathtt{x}}{\begin{array}{l}\mathtt{prod}(\mathtt{x}\,;\,\mathtt{c}(e_1{=}{<}\mathtt{x}) \star \mathtt{c}(\mathtt{x}{=}{<}e_2)\,;\,q) \rightarrow \\ \big(q[\mathtt{x}/e_1] \star q[\mathtt{x}/e_2] \star \mathtt{c}(e_1{=}{<}e_2) + \mathtt{c}(e_2{=}{<}e_1 - 1)\big)\end{array}}$$

We say that a $q$-expression is *monotone in* $x$ if $n \leq m$; then, $q[\mathtt{x}/n]$ is less than or equal to $q[\mathtt{x}/m]$. This rule does not preserve accuracy when the the c-expression is not monotone in $x$ (e.g., $\mathtt{c}(2{=}{<}x||4{=}{<}x)$).

## 4.6 Results

In the following, we present three examples that show results of programs with nested loops parameterized by an input distribution of multiple variables. The probability

distribution computed by the output program varies in complexity; the first program calculates a single parameterized output, the second program computes a triangular-shaped output distribution, and the third computes a distribution converging towards a standard normal distribution. The results are presented in a reduced and readable form extracted from our implementation.

### 4.6.1 Matrix multiplication

The original matrix multiplication program uses composite types and contains nested loops. The intermediate program, defined in Figure 4.3, contains nested recursive calls but has no dependency on data of composite types.

```
for3(i3,step,n)=
 if(i3>=n) then step else for3(i3+1,step+1,n)
for2(i2,step,n)=
 if(i2>=n) then step else for2(i2+1,for3(0,step+2,n),n)
for1(i1,step,n)=
 if(i1>=n) then step else for1(i1+1,for2(0,step,n),n)
tmulta(step,n)= for1(0,step,n)
P(step,n1)=  c(step=0)*c(n1=n)
```

Fig. 4.3: The intermediate program also containing the parameterized probability distribution. The parameter `n` is a fixed value.

The nested calls create argument development functions that depend on function calls. These are transformed into a simple form and then removed. The introduced products are over-approximated, but due to the form of the condition, the result is precise. The output program computes a single value distribution (when specialized with the size of the matrix). This is given in Figure 4.4 along with an array describing a subset of specializations of the output program with respect to a value of `n`.

```
Ptmulta(out) =
 c(3=<out/(n*n))*
 c(1=<n)*
 c(out/n*n=2+n)*1
```

| n | program |
|---|---|
| 1 | `Ptmulta(out) = c(out=3)` |
| 2 | `Ptmulta(out) = c(out=16)` |
| 3 | `Ptmulta(out) = c(out=45)` |
| 4 | `Ptmulta(out) = c(out=96)` |
| ... | ... |

Fig. 4.4: The general output probability program (left) and the program specialized with the value of `n` (right).

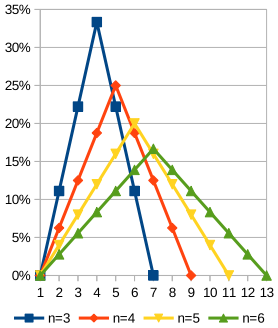
### 4.6.2 Adding parameterized distributions

This example is a recursive program computing the addition of two numbers; the input program and the input probability distribution can be seen in Figure 4.5. The output depends on both increasing and decreasing values. In this example, we use a parameter `n` as the upper limit of a range of input values. The input distribution describes two independent variables, each having a uniform distribution from 1 to $n$.

```
add(x,y) = if x=<0 then y else add(x-1,y+1)
P(x) = c(1=<x)*c(x=<n)*1/n
Pxy(x,y) = P(x)*P(y)
```

Fig. 4.5: The intermediate program containing both the `add` function and the input probability distribution. Here, the parameter `n` is used to describe a range.

The analysis gives a precise probability distribution and computes a triangular distribution (or pyramid-shaped distribution). The output probability program is described in Figure 4.6 along with a graph depicting the pyramid-shaped output probability distributions for different initializations of `n`. The lower bound on `out` arises from the input probability distribution and not from the condition. The upper bound `2*n` of the analysis result shows that the output depends on both input variables, despite the fact that one is increasing and the other is decreasing.




```
Padd(out) =
 c(2<=out)*c(out<=n)*
    (1/n*1/n*(out-1)) +
 c(1+n<=out)*c(out<=2*n)*
    (1/n*1/n*(1+2*n-out))
```

Fig. 4.6: The general output program and the graphs for the output probability distribution with n set to 3, 4, 5, and 6.

### 4.6.3 Adding 4 independent variables

The `sum4` program adds four variables and was presented by Monniaux [95]. Certain over-approximations were applied to obtain a safe and simplified result.

```
add(x,y) = if x=0 then y else add(x-1,y+1)
sum4(x,y,z,w) = add(x,add(y,add(z,w)))
tsum4(x,y,z,w) = sum4(x,y,z,w)
P(x) = c(1=<x)*c(x=<6)*1/6
Pxyzw(x,y,z,w) = P(x)*P(y)*P(z)*P(w)
```

Fig. 4.7: Intermediate program.

```
Psum4(out) =
c(4=<out)*c(out=<7)*  (-6+11*out-6*out^2 + out^3)/7776 +
c(8=<out)*c(out=<12)* (-1014+169*out+6*out^2-out^3)/7776+
c(9=<out)*c(out=<12)* (1512-461*out+42*out^2-out^3)/3888+
c(out=13)*(265/648-5*out/216)+
c(14=<out)*c(out=<18)* (-4790+923*out-54*out^2+out^3)/2592+
c(19=<out)*c(out=<24)* (17550-2027*out+78*out^2-out^3)/7776
```

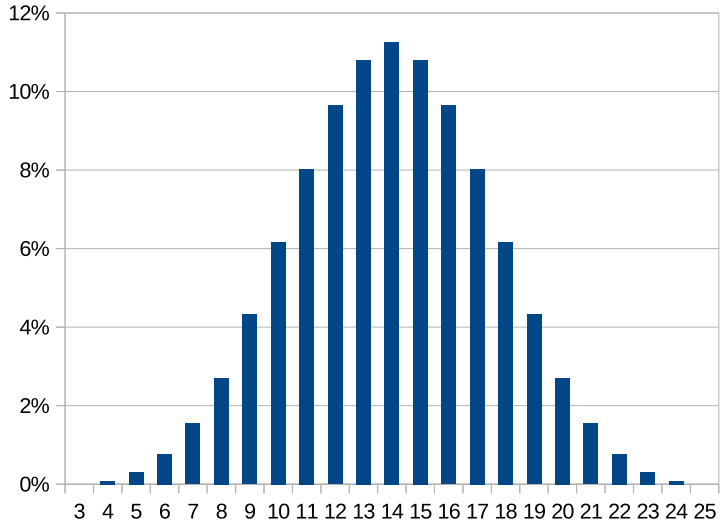



Fig. 4.8: The output program and graph for its computed probability distribution for `out` from 3 to 25.

The program is recursive, and in this example, we use independent input variables each uniformly distributed from 1 to 6, as described in Figure 4.7.

Despite the fact that the ranges and their associated value are not symmetric, the resulting program computes a precise and perfectly symmetric probability distribution, as shown in Figure 4.8. The differences in the choice of ranges comes (among other things) from the range dividing rules, as they do not divide the range symmetrically. As expected from the central limit theorem of probability theory, the resulting probability program describes a distribution that has similarities with a normal distribution.

### 4.6.4 Monty Hall

The Monty Hall problem is often used to exemplify how gained knowledge influences probabilities (conditional probability). In this problem, there are three closed doors, one hiding a prize and two that are empty. The doors have an equal chance of hiding the prize. There is a contestant who chooses one of the doors; then, the game host will open an empty door, and the contestant can either stick with the first choice or change to the other unopened door. The problem lies in determining whether the best winning-strategy is to stick with the first choice or to switch to the other.

If the strategy is to stick with the first choice and that door has a prize, then the contestant has won. If the contestant changes doors, he/she only loses if the first choice was the door hiding the prize; if the first choice was an empty door, then the game host would open the other empty door, leaving only the prize door as a second choice.

The `monty` program models the two strategies: If the strategy variable is 1, then the strategy is to change the door; otherwise, the strategy is to stick with the first choice. The program takes as input the contestant's first guess, the door hiding the prize, the empty door, which is not opened by the game host, and the strategy that the contestant uses.

Let us assume that the contestant has an equal chance of choosing each of the doors. The input variables `guess`, `price`, and `empty` model the first choice, the prize door and the empty door, which is left after the game host has opened an empty door. All three doors have a value between 1 and 3, and the empty door cannot be the same as the prize door. We have parametrized the strategy with a weight `p` between the two such that when `p` = 1, the strategy is to always change doors, and when `p`=0, the strategy is to always keep the first choice (e.g., letting `p` = 0.75, we change doors in 3/4 of cases and keep the first door in the remaining cases). Such a parametrization allows us to execute the analysis once and use the lighter closed-form result for that calculation instead. In a problem whereby the winning probability of a strategy is dependent on the other input, such input could be used for optimizing the choice of strategy. The `monty` program and the parametrized input probability distribution can be seen in Figure 4.9.

The analysis was capable of handling the program correctly, and the result can be seen in Figure 4.10. The probabilities 1/3 and 2/3 do not occur directly in the output probability program but rather are found in the constants 6, 12 and 1/18.

### 4.6.5 Adding dependent non-uniform variables

A function call may have interdependent and non-uniform arguments, and in this example, we demonstrate that the analysis can handle such function calls. We focus on the dependencies, analyze a simple add program and discuss the limits of the interdependencies. The program also shows that interdependencies quickly lead to the occurrence of integer division in the output

The input arguments are interdependent; the second argument is always less than or equal to the value of the first argument. The joint distribution depends only on

```
monty(guess,price,empty,strategy)=
  if strategy = 0 then finalGuess(guess,price)
                  else change(guess,price,empty)

finalGuess(guess,price)=
  if price=guess then 1 else 0

change(guess,price,empty)=
  if price=guess then finalGuess(empty,price)
                 else finalGuess(price,price)

Pin(guess,price,empty,strategy) =
  1/18*c(1=<guess)*c(guess=<3)*
       c(1=<price)*c(price=<3)*
       c(1=<empty)*c(empty=<3)*
       c(not(price = empty))*
       Pstrat(strategy)

Pstrat(strategy) =
  p*c(1=strategy) + (1-p)*c(0=strategy)
```

Fig. 4.9: The `monty` program models the event flow depending on the chosen strategy; if the strategy is `0`, then the contestant keeps the first door, and if it is `1`, then the contestant changes his mind. There are three doors, and the input of `monty` describes the contestant's first guess, the door hiding the prize, the empty door that is not opened by the game host (and is different from the prize door), and the strategy of the contestant. If the final choice hides the prize, then the program returns 1; otherwise, it returns 0. The probability of the strategy is an expression parametrized with a weight `p` between the two strategies instead of executing the analysis twice with different strategies.

```
pmonty(out) =
 1/18 *
 (c(out=0)*
   (12*(1-p)+6*p)+
  c(out=1)*
   (6*(1-p)+12*p))
```

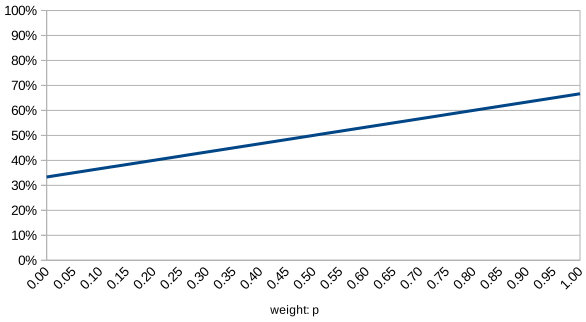



Fig. 4.10: The probability of winning the Monty Hall as a function of the weight given to the change strategy. The probabilistic output analysis reveals that the best weight between the keep strategy and the change strategy is to always use the change strategy.

the value of the first argument, resulting in a skewed probability distribution. The probability program is defined in Figure 4.11.

```
Pxy(x,y) =  c(1=<y)*c(y=<3) * c(1=<x)*c(x=<y) * x/10

add(x,y) = x+z

Padd(out) =
  c(2=<out)*c(out=<3)*  1/20 *out%2*(1 + out%2) +
  c(4=<out)*c(out=<6)*-(1/20)*(-4+out-out%2)*(-3+out+out%2)
```

Fig. 4.11: An input program, `add`; its skewed joint distribution, `Pxy`; and the closed-form probability program, `Padd`; produced by the analysis. The integer division is noted by a "`%`".

The create rule generates nested summations, and removing such inner summations implies that their values must be expressed using the variables of the outer summations or the input variable (i.e., `out`). Comparing the result from this experiment with the output probability distribution for the addition of two random variables in Figure 4.6 indicates that integer division is a special case arising from a dependent input. The following interesting expressions are extracted during analysis execution, and they show how the integer division arises from the dependency of the input. The first expressions are the result of the create rule, and the last expression is the result after the removal of the inner y-summation.

$$
\begin{aligned}
&P_{\texttt{add}}(\texttt{out}) = \\
&\texttt{sum}(\texttt{x}\,;\texttt{sum}(\texttt{y}\,;\texttt{c}(\texttt{out}=\texttt{x}+\texttt{y})*\\
&\quad \texttt{c}(1=<\texttt{x})*\texttt{c}(\texttt{x}=<\texttt{y})*\texttt{c}(1=<\texttt{y})*\texttt{c}(\texttt{y}=<3)*(\texttt{i2r}(\texttt{x})\;/\;\texttt{i2r}(10)))) =
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{sum}(\texttt{x}\,;\texttt{c}(2=<\texttt{out})*\texttt{c}(\texttt{out}=<3)*\texttt{c}(1=<\texttt{x})*\\
&\quad \texttt{c}(2*\texttt{x}=<\texttt{out})*(\texttt{i2r}(\texttt{x})\;/\;\texttt{i2r}(10)))+\\
&\texttt{sum}(\texttt{x}\,;\texttt{c}(4=<\texttt{out})*\texttt{c}(\texttt{out}=<3+\texttt{x})*\texttt{c}(2*\texttt{x}=<\texttt{out})*(\texttt{i2r}(\texttt{x})\;/\;\texttt{i2r}(10)))
\end{aligned}
$$

In the last expression, there are two summations, each leading to their own part in the resulting program. Looking closely at each summation, we see that they share the upper limit for `x`, $\texttt{c}(2*\texttt{x}=<\texttt{out})$, which currently contains an integer multiplication and when solved with respect to `x` contains the integer division. In the final result, the second part of the expression has an upper limit on `out`, $\texttt{c}(\texttt{out}=<6)$, which is a constraint that the summation-removal rule introduces to ensure that the lower limit of the summation (i.e., $\texttt{out}-3$) is less than or equal to the upper limit (i.e., $\texttt{out}\,\%\,2$).

The original probability $(\texttt{i2r}(\texttt{x})\;/\;\texttt{i2r}(10))$ occurs directly in the summations, and this indicates a limit of this implementation and approach. To be able to handle a probability, the rewrite rules for summations must transform summations

over the probability expression. There are limits to which series that can be transformed currently; for instance, the sum of reciprocals (e.g., $\sum_{k=1}^{n} \frac{1}{k}$), known as a harmonic series and variations hereof, such as generalized harmonic series, cannot be handled by the current implementation. The current analysis is limited to finite summations of at least order of 1, but a more extensive use of Mathematica [146] that exploits more of Mathematica's rewriting mechanisms may be able to handle such series.

## 4.7 Related works

Probabilistic analysis is related to the analysis of probabilistic programs. Probabilistic analysis is the analysis of programs with a normal semantics wherein the input variables are interpreted over probability distributions. The analysis of probabilistic programs analyses programs with probabilistic semantics wherein the values of the input variables are unknown (e.g., flow analysis [108]).

In probabilistic analysis, it is important to determine how variables depend on each other; however, already in 1976, Denning had proposed a flow analysis for revealing whether variables depend on each other [36]. This was presented in the field of secure flow analysis. Denning introduced a lattice-based analysis wherein she, given the name of a variable, which should be kept secret, deduced which other variables should be kept secret to avoid the leaking of information. In 1996, Denning's method was refined by Volpano *et al*. into a type system, and for the first time, it was proven sound [139].

Reasoning about probabilistic semantics is an area closely related to probabilistic analysis, as they both work with nested probabilistic influences. Probabilistic analysis works on standard semantics and analyses them using input probability distributions, where a probabilistic semantics allow for random assignments and probabilistic choices [83] and is normally analysed using an expanded classical analysis or verification method [33].

Probabilistic model checking is an automated technique for formally verifying quantitative properties for systems with probabilistic behaviors. It is mainly focused on Markov decision processes, which can model both stochastic and nondeterministic behaviours [52, 84]. This differs from probabilistic analysis in that it assumes the Markov property.

In 2000, Monniaux applied abstract interpretation to programs with probabilistic semantics and determined safe bounds for worst-case analysis [95]. Pierro *et al*. introduced a linear mapping structure, a Moore-Penrose pseudo-inverse, instead of a Galois connection. They used the linear structures to compare the 'closeness' of approximations as an expression using the average approximation error. Pierro *et al*. further explored using probabilistic abstract interpretation to calculate the average case analysis [40]. In 2012, Cousot and Monerau gave a general probabilistic abstraction framework [33] and stated, in section 5.3, that Pierro *et al*.'s method and many other abstraction methods can be expressed in this new framework.

When analysing probabilities, the main challenge is to maintain the dependencies throughout the program. Schellekens defines this as *Randomness preservation* [124] (or random bag preservation), which in his (and Gao's [57]) case enables the tracking of certain data structures and their distributions. They use special data structures, as they found these suitable to derive the average number of basic operations. In another approach [109, 141], tests in programs have been assumed to be independent of previous history, also known as the Markov property (the probability of true is fixed). As Wegbreit remarked, this is true only for certain programs (e.g., linear search for repeating lists); for other programs, this is not the case (linear search for non-repeating lists). The Markov property is the foundation of Markov decision processes, which are used in probabilistic model checking [52]. Cousot et al. presented a probabilistic abstraction framework wherein they divide the program semantics into probabilistic behaviours and (non-)deterministic behaviours. They proposed the handling of tests when it is possible to assume the Markov property and the handling of loops using a probability distribution describing the probability of entering the loop in the $i$th iteration. Monniaux proposed another approach for abstracting probabilistic semantics [95]; he first lifts a normal semantics to a probabilistic semantics whereby random generators are allowed and then uses an abstraction to reach a closed form. Monniaux's semantic approach uses a backward probabilistic semantics operating on measurable functions. This is closely related to the forward probabilistic semantics proposed earlier by Kozen [83].

An alternative approach to probabilistic analysis is based on the symbolic execution of programs with symbolic values [58]. Such techniques can also be used on programs with infinitely many execution paths by limiting the analysis to a finite set of paths at the expense of the tightness of probability intervals [120].

## 4.8 Conclusion

The probabilistic analysis of program has seen renewed interest for analysing programs with respect to energy consumptions. Numerous embedded systems and mobile applications are limited by restricted battery life. In this paper, we describe a rewrite system that derives a resource probability distribution for programs given distributions of inputs. The system can analyse programs in a subset of C where we have known distributions of input variables. From the original program, we create a probability distribution program, where we remove calls to original functions and transform them into closed form. We have presented the transformation rules for each step and outlined the implementation of the system. We discuss over-approximating rules and their influence on the accuracy of the output probability, and we show that our analysis improves on related analysis in the literature.

## 4.9 Afterword

In this chapter, we have presented a novel technique for probabilistic resource analysis whereby the analysis is seen as a program-to-program transformation. The anal-

ysis was evaluated on a series of small programs, for which it was able to find accurate probability distributions. The implementation of the analysis was naively constructed which is reflected in its execution times. For instance, the implementation spent an hour analysing the `sum4`-program; the majority of that hour was spent opening and closing the connection to Mathematica. A re-implementation using Mathematica would benefit from being implemented in C; for which there exists a library providing direct access to Mathematica operations on C objects.

A missing related work is an article by Burstall and Darlington [22], who presents a transformation-based system that transforms recursive programs in the form of recurrence equations into equivalent and perhaps more efficient recursive programs. By equivalent, we mean a program that has the same input-output relation. Our system is not intended to preserve the input-output relation, and whereas they may produce recursive programs, we are interested in a closed-form expression. However, the above rules could perhaps be reformulated using the easy-to-comprehend notation that they use.

The set of C programs the analysis handles well is limited by (i) the discrete resource model, e.g., the utilized resource model has no dependencies between the resource usages; (ii) the program transformation from C to the functional representation; and (iii) the capabilities of the probabilistic analysis. Since the underlying probabilistic analysis cannot handle lists or arrays, the resource analysis can only handle programs with arrays or lists if the resource model and slicing remove them. Furthermore, the transformation must be able to translate the sliced program into the functional representation, and the probabilistic analysis must be capable of transforming that program into a closed form. The three aspects together cloud the picture of which C programs can be handled.

Since the underlying probabilistic analysis is not able to handle lists or other non-primitive types, the overall analysis can only handle programs with *e.g.* lists if these are removed by the resource model and slicing. This is a challenge that we will solve implicitly in the approaches presented in the following chapter; when employing an existing analysis that handles such types, both approaches may infer probability bounds for programs with such types.

The above presented rules where sufficient to analyse the test programs, however, more rules would be needed for more complex programs: in the following cases, the technique returns trivial bounds (i) when the body of a loop is reduced to a recurrence in-equations instead of recurrence equations, *e.g.*, see Figure 4.12a, since there is only a rewrite rule, *i.e.*, “rem(ArgDev)”, for removing `argDev`, and (ii) when the function arguments are interdependent since neither can be reduced to a form whereby the “rem(ArgDev)”-rule can be applied, *e.g.*, see Figure 4.12b. These problems may be encountered using the techniques of cost analysis [7, 127].

```
f(i,y) = if i>=2 then y else f(i+1, g(y))
g(y)   = if (*) then y+1 else y
```

(a)

```
dep(x,y) = if y<=1 then x else dep(y-1,x-1)
```

(b)

Fig. 4.12: (a): A program where * indicates that the we do not know whether the test evaluates to true or false. Thus, we cannot derive an accurate solution for the loop body. (b): A program where the argument values are dependent.

# 5

# Probabilistic Output Measures based on existing analyses

In the previous chapters, we showed how a discrete probability distribution and a program can be used for deriving probability bounds for output events and discrete resource usages. In this chapter, we take a step back and return to the general case whereby the input is described by a probability measure.

Recall that a program semantics $|\texttt{prg}|$ is considered to be a relation between input $X$ and output $Y$, that is, $|\texttt{prg}| \subseteq X \times Y$. In this chapter, we prove that we may derive probability bounds for output events from an over-approximation $pre^{\sharp}_{|\texttt{prg}|} \colon \wp(Y) \to \wp(X)$ of the pre-image $pre_{|\texttt{prg}|} \colon \wp(Y) \to \wp(X)$, *i.e.*, $pre_{|\texttt{prg}|}(A) \subseteq pre^{\sharp}_{|\texttt{prg}|}(A)$, and a probability measure over input events.

This chapter is based on the idea of "reusing existing analyses", either forward or backwards, to obtain the over-approximation of the pre-image. We consider an analysis to be given as a function $pre^{\sharp}_{|\texttt{prg}|} \colon \wp(Y) \to \wp(X)$ (backwards) or a function $img^{\sharp}_{|\texttt{prg}|} \colon \wp(X) \to \wp(Y)$ (forwards) that over-approximates the concrete $pre_{|\texttt{prg}|}$ and $img_{|\texttt{prg}|}$, respectively. An analysis $|\texttt{prg}|^{\sharp}$ is typically given in some abstract domain using an abstraction from the concrete domain to the abstract domain, but we assume a concretization function to avoid complications of the abstract domain. For instance, a forward interval analysis is actually a function $|\texttt{prg}|^{\sharp} \colon \mathsf{Interval} \to \mathsf{Interval}$ rather than $img^{\sharp}_{|\texttt{prg}|} \colon \wp(\mathcal{R}) \to \wp(\mathcal{R})$; however, we assume that we compose with the concretization function $\gamma \colon \mathsf{Interval} \to \wp(\mathcal{R})$ and the abstraction function $\alpha \colon \wp(\mathcal{R}) \to \mathsf{Interval}$, that is, $img^{\sharp}_{|\texttt{prg}|} = \gamma \circ |\texttt{prg}|^{\sharp} \circ \alpha$.

*Foreword.* The content of this chapter is unpublished.

## 5.1 Preliminaries

We refer to Appendix A for the basic definitions of relations and functions and their properties and to Chapter 2.2 for the preliminary results of inducing probabilities via functions.

## 5.2 Backwards analysis

Let a program $\texttt{prg}$ from input $X$ to output $Y$ have a semantics $|\texttt{prg}|\colon X \to Y$ that is a measurable function, *i.e.*, $|\texttt{prg}|\colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$. Given an input probability measure $\mu\colon \mathcal{X} \to [0,1]$, the probability of an output event $A \in \mathcal{Y}$ is defined as the input probability of its pre-image $pre_{|\texttt{prg}|}(A)$, namely, $\mu(pre_{|\texttt{prg}|}(A))$. If the $|\texttt{prg}|$ is clear from the context, it will be omitted.

In this section, we do not have $pre(A)$; instead, we have a pre-image over-approximating backwards analysis, *e.g.* [31], that is a function $pre^\sharp\colon \wp(Y) \to \wp(X)$ such that $pre(A) \subseteq pre^\sharp(A)$. We want to use $pre^\sharp$ to provide upper and lower probability bounds for all output events. This presents us with some challenges.

A summary of the approach is as follows. First, we must ensure that $pre^\sharp(A)$ is a measurable input event when $A$ is (a measurable) output event. We construct a new function $pre'^\sharp$, where $pre'^\sharp(A)$ is a (measurable) input event, *i.e.*, $pre'^\sharp(A) \in \mathcal{X}$. To do this, we define a new function $\uparrow\colon \wp(X) \to \mathcal{X}$, which is increasing, *i.e.* $B \subseteq \uparrow B$. Then, we define $pre'^\sharp(A) \triangleq \uparrow pre^\sharp(A)$, which produces measurable input events for all subsets of the output and specifically for the measurable output events. Second, we need to define a dual function $pre'^\flat\colon \wp(Y) \to \wp(X)$ that under-approximates the pre-image, *i.e.* $pre(A) \supseteq pre'^\flat(A)$. We use $pre'^\sharp$ to define $pre'^\flat$ and show that if $pre'^\sharp(A)$ is a measurable input event, then $pre'^\flat(A)$ is as well (and vice versa). Now, we are able to define upper and lower probability bounds (Theorem 5.10). Afterwards, we discuss how to choose $\uparrow$ to achieve the tightest possible bounds and when such a best $\uparrow$ exists.

We start by defining an order between the pre-image functions based on the relationship of their outputs.

**Definition 5.1.** *Let $pre, pre^\sharp, pre^\flat\colon \wp(Y) \to \wp(X)$ be functions. The function $pre^\sharp$* over-approximates *$pre$,* i.e. *$pre \preceq pre^\sharp$, if $pre(A) \subseteq pre^\sharp(A)$. The function $pre^\flat$* under-approximates *$pre$,* i.e. *$pre^\flat \preceq pre$, if $pre^\flat(A) \subseteq pre(A)$.*

Now, we introduce the concept of a dual function, which we can use to define such under-approximating pre-images based on over-approximating pre-images. The duality also provides a useful relationship between the upper and lower probability bounds; see *e.g.* Theorem 5.10.

**Definition 5.2.** *Let $f\colon \wp(Y) \to \wp(X)$ be a function. A function $\widetilde{f}\colon \wp(Y) \to \wp(X)$ is* dual of *$f$ if $\widetilde{f}(A) = f(A^{\complement})^{\complement}$.*

Because $|\texttt{prg}|$ is total, we can use the *dual* of $pre^\sharp$ to define a function $pre^\flat$ that under-approximates $pre$, as shown by the following lemma.

**Lemma 5.3.** *Let $pre_f, pre_f^\sharp\colon \wp(Y) \to \wp(X)$ be functions such that $pre_f$ is the pre-image of a measurable function $f\colon X \to Y$ and $pre_f \preceq pre_f^\sharp$. Then, the dual $pre_f^\flat \triangleq pre_f^\sharp(A^{\complement})^{\complement}$ under-approximates the pre-image $pre_f$, that is,*

$$pre_f^\flat \preceq pre_f.$$

*Proof.* Since $f$ is total, $pre_f(A) \cup pre_f(A^{\complement}) = X$. Thus,

$$\begin{aligned} pre_f^{\flat}(A) = pre_f^{\sharp}(A^{\complement})^{\complement} = X \setminus pre_f^{\sharp}(A^{\complement}) &= \Big(pre_f(A) \cup pre_f(A^{\complement})\Big) \setminus pre_f^{\sharp}(A^{\complement}) \\ &= pre_f(A) \setminus pre_f^{\sharp}(A^{\complement}) \\ &\subseteq pre_f(A). \end{aligned}$$

In the literature, another definition of dual functions is sometimes used, namely, $\widetilde{f}(A) = f(A) \setminus f(A^{\complement})$; in our case, they are equivalent.

**Proposition 5.4.**

$$pre_f^{\flat}(A) = pre_f^{\sharp}(A) \setminus pre_f^{\sharp}(A^{\complement})$$

*Proof.* We first prove that $pre_f^{\sharp}(B^{\complement})^{\complement} \subseteq pre_f^{\sharp}(B)$: we obtain $pre_f^{\sharp}(Y) = X$ by the assumption $pre_f(Y) \subseteq pre_f^{\sharp}(Y)$ and $pre_f(Y) = X$ because $f$ is total. Thus, $pre_f^{\sharp}(B^{\complement})^{\complement} = X \setminus pre_f^{\sharp}(B^{\complement}) = pre_f^{\sharp}(Y) \setminus pre_f^{\sharp}(B^{\complement}) = \Big(pre_f^{\sharp}(B) \cup pre_f^{\sharp}(B^{\complement})\Big) \setminus pre_f^{\sharp}(B^{\complement}) = pre_f^{\sharp}(B) \setminus pre_f^{\sharp}(B^{\complement}) \subseteq pre_f^{\sharp}(B)$. Based on that, we obtain $pre^{\flat}(B) = pre_f^{\sharp}(B) \setminus pre_f^{\sharp}(B^{\complement}) = pre_f^{\sharp}(B) \cap pre_f^{\sharp}(B^{\complement})^{\complement} = pre_f^{\sharp}(B^{\complement})^{\complement} = \widetilde{pre_f^{\sharp}}$.

Monotonicity[1] becomes interesting later when we create upper and lower probability bounds because this ensures that the bounds also become monotonic, as with the probability measures.

**Lemma 5.5.** *If $pre_f^{\sharp}$ is monotone then $pre_f^{\flat}$ is monotone.*

*Proof.* Assume $A, B \subseteq X$, where $A \subseteq B$ and define $C = (B \setminus A)$; then, $pre_f^{\flat}(B) = pre_f^{\flat}(A \uplus C) = pre_f^{\sharp}(A \uplus C) \setminus pre_f^{\sharp}((A \uplus C)^{\complement}) = pre_f^{\sharp}(A \uplus C) \setminus pre_f^{\sharp}(A^{\complement} \cap C^{\complement})) \supseteq pre_f^{\sharp}(A \uplus C) \setminus pre_f^{\sharp}(A^{\complement}) \supseteq pre_f^{\sharp}(A) \setminus pre_f^{\sharp}(A^{\complement}) = pre_f^{\flat}(A)$

The intention is to measure the over-approximated and under-approximated pre-images of each output event $A$ using the input probability measure $\mu \colon \mathcal{X} \to [0, 1]$. However, we can only do this if $pre^{\sharp}(A)$ and $pre^{\flat}(A)$ both exist in $\mathcal{X}$. However, this is not always the case, as shown in the following example.

*Example 5.6.* Let $f \colon (\{a, b\}, \{\emptyset, \{a, b\}\}) \to (\{c, d\}, \wp(\{b, c\})$ be a measurable function whereby $f(a) = f(b) = c$ (so that $pre_f(\{d\}) = \emptyset$), and let the pre-image over-approximating function $pre^{\sharp}$ be defined such that $pre_f^{\sharp}(\{d\}) = \{b\}$. Here, $pre_f^{\sharp}(\{d\}) \notin \{\emptyset, \{a, b\}\}$.

For these cases, we construct a new function that further over-approximates the pre-images using an abstraction function $\uparrow$.

**Definition 5.7.** *Let $(X, \mathcal{X})$ be a measurable space. A function $\uparrow \colon \wp(X) \to \mathcal{X}$ is an* abstraction *if $A \subseteq \uparrow A$.*

[1] A function $f \colon \wp(X) \to \wp(Y)$ is monotone if $\forall A, B \in \wp(X), A \subseteq B \Rightarrow f(A) \subseteq f(B)$.

Such an abstraction $\uparrow\colon \wp(X) \to \mathcal{X}$ can always be defined given a mapping $f\colon X \to \mathcal{X}$, where $x \in f(x)$, *i.e.* $\uparrow B \triangleq \bigcup_{b \in B} f(b)$.

The composition $pre'^{\sharp}$ of a pre-image over-approximation $pre^{\sharp}$ and an abstraction $\uparrow$ both over-approximates the pre-image and produces measurable input events.

**Lemma 5.8.** *Let* $pre\colon \wp(Y) \to \wp(X)$ *be a pre-image, let* $pre^{\sharp}$ *be its over-approximation,* i.e. $pre \preceq pre^{\sharp}$*, and let* $\uparrow\colon \wp(X) \to \mathcal{X}$ *be an abstraction; then,*

$$pre \preceq \uparrow \circ pre^{\sharp} \text{ and } \uparrow pre^{\sharp}(A) \in \mathcal{X}$$

*Proof.* The proof is trivial using the definitions of $\uparrow$ and $\preceq$.

When the over-approximated pre-images of the output events are measurable in the input measure space, their dual under-approximated pre-images are also measurable in the input space, as the following lemma states.

**Lemma 5.9.** *Let* $f\colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$ *be a measurable function, let* $pre_f^{\sharp}\colon \wp(X) \to \wp(Y)$ *be a function whereby* $pre_f \preceq pre_f^{\sharp}$*, and let* $pre_f^{\flat}$ *be the dual of* $pre_f^{\sharp}$*. Then, for all* $A \in \mathcal{Y}$*,*

$$pre_f^{\sharp}(A) \in \mathcal{X} \text{ if and only if } pre_f^{\flat}(A) \in \mathcal{X}$$

*Proof.* Let $A \in \mathcal{Y}$. The following are consequences of $\sigma$-algebras being closed under complements, of the duality of $pre_f^{\flat}$ and $pre^{\sharp}$, and of the assumed measurability of $A$.

"$\Rightarrow$": $A \in \mathcal{Y} \Rightarrow A^{\complement} \in \mathcal{Y} \Rightarrow pre_f^{\sharp}(A^{\complement}) \in \mathcal{X} \Rightarrow pre_f^{\sharp}(A^{\complement})^{\complement} \in \mathcal{X} \Rightarrow pre_f^{\flat}(A) \in \mathcal{X}$

"$\Leftarrow$": $A \in \mathcal{Y} \Rightarrow A^{\complement} \in \mathcal{Y} \Rightarrow pre_f^{\flat}(A^{\complement}) \in \mathcal{X} \Rightarrow pre_f^{\sharp}(A^{\complement\complement})^{\complement} \in \mathcal{X} \Rightarrow pre_f^{\sharp}(A)^{\complement} \in \mathcal{X} \Rightarrow pre_f^{\sharp}(A) \in \mathcal{X}$

This concludes the series of lemmas, and we can now present the first theorem providing upper and lower probability bounds for all output events.

**Theorem 5.10.** *Let* $f\colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$ *be a measurable function, let* $(X, \mathcal{X}, \mu)$ *be an input probability space, and let* $\mu_f\colon \mathcal{Y} \to [0, 1]$ *be the output probability measure,* i.e. $\mu_f = \mu \circ pre_f$*. Furthermore, let* $pre_f^{\sharp}\colon \wp(Y) \to \wp(X)$ *over-approximate* $pre_f$*,* i.e. $pre_f \preceq pre_f^{\sharp}$*, and let* $\uparrow\colon \wp(X) \to \mathcal{X}$ *be an abstraction. We let* $pre'^{\sharp}_f \triangleq \uparrow pre_f^{\sharp}$ *and* $pre'^{\flat}_f(A) \triangleq pre'^{\sharp}_f(A^{\complement})^{\complement}$*, and we define* upper $\mu^{\sharp}$ and lower $\mu^{\flat}$ probability bounds of $\mu_f$ *as* $\mu_f^{\flat} \triangleq \mu \circ pre'^{\flat}_f$ *and* $\mu_f^{\sharp} \triangleq \mu \circ pre'^{\sharp}_f$*. Then,*

$$\mu_f^{\flat}(A) \leq \mu_f(A) \leq \mu_f^{\sharp}(A)$$

*and*

$$\mu_f^{\flat}(A) = 1 - \mu_f^{\sharp}(A^{\complement})$$

*Furthermore, if* $pre_f^{\sharp}$ *and* $\uparrow$ *are monotonic, then* $\mu_f^{\sharp}$ *and* $\mu_f^{\flat}$ *are monotonic.*

*Proof.* By Lemmas 5.8 and 5.9, $pre'^{\sharp}_f(A), pre'^{\flat}_f(A) \in \mathcal{X}$ and $pre'^{\flat}_f \preceq pre_f \preceq pre'^{\sharp}_f$. Furthermore, by the monotonicity of $\mu$, we obtain $\mu(pre'^{\flat}_f(A)) \leq \mu(pre_f(A)) \leq \mu(pre'^{\sharp}_f(A)$. Thus, $\mu^{\flat}_f(A) \leq \mu_f(A) \leq \mu^{\sharp}_f(A)$. We obtain the second part by Proposition 2.16, *i.e.*, $\mu(A) = 1-\mu(A^{\complement})$ and the definitions of $\mu^{\flat}_f$ and $\mu^{\sharp}_f$, that is, $\mu^{\flat}_f(A) = \mu(pre'^{\flat}_f(A)) = 1 - \mu(pre'^{\flat}_f(A)^{\complement}) = 1 - \mu(pre'^{\sharp}_f(A^{\complement})) = 1 - \mu^{\sharp}_f(A^{\complement})$. Finally, since $\mu$, $pre^{\sharp}_f$ and $\alpha$ are monotonic, then by composition and Lemma 5.5, $\mu \circ pre'^{\sharp}_f$ and $\mu \circ pre'^{\flat}_f$ are monotonic.

### 5.2.1 Precision of probability bounds

We are clearly interested in the tightest possible probability bounds. In some cases, we can find the best possible bound, but this does not hold in general. To achieve the tightest probability bounds, the abstraction $\uparrow A$ should return not only some increased element that is in the $\sigma$-algebra $\mathcal{X}$ but also the least element of those. However, such a least element does not necessarily exist.

**Lemma 5.11.** *Let $(X, \mathcal{X})$ be a measurable set, and let $A \in \wp(X)$; there does not always exist a least $B \in \mathcal{X}$ such that $A \subseteq B$.*

*Proof. Proof by counterexample.* A set $A \subseteq X$ is *co-countable* if $A^{\complement}$ is countable. We define a $\sigma$-algebra $\mathcal{X}$ to be that generated by the collection of all countable and co-countable subsets of $X$. Note that since each singleton set is countable, they all exist in $\mathcal{X}$.

Now, let $A \in \wp(X)$ be uncountable with an uncountable complement $A^{\complement}$. We will show (by contradiction) that there is no least $B \in \mathcal{X}$ such that $A \subseteq B$. Assume that there is a least set $B \in \mathcal{X}$ that contains $A$. Then, $B$ would need to be uncountable, and according to the definition of $\mathcal{X}$, $B^{\complement}$ would be countable. Since $B^{\complement}$ is countable and $A^{\complement}$ is uncountable, $B^{\complement} \subset A^{\complement}$. This is equivalent to $A \subset B$, which causes $B \setminus A$ to contain at least one element; let that element be $x$. Because $\{x\}$ is a singleton set, $\{x\} \in \mathcal{X}$, and because $\mathcal{X}$ is closed under countable intersection, $B \setminus \{x\} \in \mathcal{X}$. This implies that there is another set, namely, $B \setminus \{x\}$, such that $A \subseteq B \setminus \{x\} \subset B$, and thus $B$ is not the least set in $\mathcal{X}$ that contains $A$ - this contradicts our assumption.

In some cases where the $\sigma$-algebra is a complete lattice, such a least element does exist. In the following, we will provide two cases wherein the $\sigma$-algebras are complete lattices; we will start with the easy case, namely, power sets.

**Lemma 5.12.** *A power set $\wp(X)$ of a set $X$ is a $\sigma$-algebra.*

**Lemma 5.13.** *A power set $\wp(X)$ of a set $X$ is a complete lattice.*

If the $\sigma$-algebra is a complete lattice, then the abstraction is the identity function, *i.e.* $\uparrow = id$. In the second case, we will use a special $\sigma$-algebra constructed from a countable partition over the inputs.

**Definition 5.14.** *A* partition $T$ *of a set* $X$ *is a set of disjoint subsets of* $X$*, where* $\emptyset \notin T$ *and* $\bigcup T = X$*. If* $T$ *is a finite/countable/infinite set, we say that* $T$ *is a* finite/countable/infinite partition*.*

In contrast to case one, the following is more general in that the $\sigma$-algebra need not be a power set and is more limited in that the partition it uses must be countable.

**Theorem 5.15.** *Let* $X$ *be a set, and let* $T$ *be a countable partition of* $X$*. Let* $(X, \sigma(T))$ *be a measurable space, and let* $A \subseteq X$*; then, there exists a least element* $B$ *in* $\mathcal{X}$ *such that* $A \subseteq B$*.*

*Proof.* Let $I \subseteq \mathbb{N}$ be a set with as many elements as $T$, *i.e.* $|T| = |I|$. Then, $\sigma(T)$ is isomorphic to $\wp(I)$ [2]. Since such a power set is a complete lattice (By Lemma 5.13), $\mathcal{X}$ is as well. Given $A$, we define $B = \bigcap\{C \in \mathcal{X} | A \subseteq C\}$, where $A$ is clearly a subset of $B$, and *(a)* $B$ is in $\mathcal{X}$, and *(b)* it is the least because $\mathcal{X}$ is a complete lattice.

The important point in the above theorem is that the partition is countable; in general, a $\sigma$-algebra is not a complete lattice, as indicated in Lemma 5.11. For instance, in the $\mathcal{B}(\mathbb{R})$ algebra, every singleton set of $\mathbb{R}$ exists in $\mathcal{B}(\mathbb{R})$. If a $\mathcal{B}(\mathbb{R})$ was a complete lattice, this would imply that all subsets of $\mathbb{R}$ are in $\mathcal{B}(\mathbb{R})$. However, the cardinalities are different, that is, $|\mathcal{B}(\mathbb{R})| = 2^{\aleph_0}$ and $|\wp(\mathbb{R})| = 2^{2^{\aleph_0}}$ [134], which indicates that not all subsets of $\mathbb{R}$ are in $\mathcal{B}(\mathbb{R})$. Thus, $\mathcal{B}(\mathbb{R})$ is not a complete lattice.

## 5.3 Forward analysis

Again, let $|\texttt{prg}|\colon X \to Y$ be a function, and recall that $img_f(A) \triangleq \{f(x) \mid x \in A\}$. In this section, we present a method for computing upper and lower probability bounds for output events provided a probability measure $\mu\colon \mathcal{X} \to [0,1]$ over the input $X$ and a reusable forward analysis, that is, a computable over-approximation $img^\sharp_{|\texttt{prg}|}\colon \wp(X) \to \wp(Y)$ of the image-function $img_{|\texttt{prg}|}\colon \wp(X) \to \wp(Y)$, *i.e.*, $img_{|\texttt{prg}|} \preceq img^\sharp_{|\texttt{prg}|}$. To compute the probability of the events, we only need to define a computable pre-image over-approximating function $pre^\sharp_f$, and then, we can apply Theorem 5.10 and obtain Theorem 5.15.

We may define the pre-image function based on the image function since the image function on the singletons defines the program semantics by definition.

**Lemma 5.16.** *Let* $f$ *be a function with image-function* $img_f\colon \wp(X) \to \wp(Y)$*; then,*

$$pre_f(A) = \{x \in X \mid img_f(\{x\}) \cap A \neq \emptyset\}.$$

*Proof.* When $f$ is a function, $img_f(\{x\}) = \{f(x)\}$. Thus, the above is a direct consequence of the definition of $pre_f$, *i.e.* $pre_f(A) \triangleq \{x \in X \mid f(x) \in A\}$.

[2] Two partially ordered sets $X$ and $Y$ are isomorphic if there exist an order-preserving bijection $f\colon X \to Y$ with an existing inverse $f^{-1}$ that is also order preserving.

In our case, we do not have an image function; rather, we have an over-approximation of the image-function $img^\sharp_f$, *i.e.*, $img_f \preceq img^\sharp_f$. We may instead use that to define an over-approximation of the pre-image function $pre^\sharp_f$.

**Lemma 5.17.** *Let $f$ be a function with the image function $img_f \colon \wp(X) \to \wp(Y)$ and pre-image function $pre_f$, and let $img^\sharp_f$ be a function whereby $img_f \preceq img^\sharp_f$. If we let $pre^\sharp_f(A) \triangleq \{x \in X \mid img^\sharp_f(\{x\}) \cap A \neq \emptyset\}$, then $pre^\sharp_f(A) \supseteq pre_f(A)$.*

*Proof.* $pre_f(A) = \{x \in X \mid img_f(\{x\}) \cap A \neq \emptyset\} \subseteq \{x \in X \mid img^\sharp_f(\{x\}) \cap A \neq \emptyset\} = pre^\sharp_f(A)$.

For the subclass whereby $X$ is finite, any event $A \in \wp(Y)$ is computable. However, when $X$ is infinite, the pre-images are uncomputable, *i.e.* they require infinitely many tests/computations. An exception is the trivial $\sigma$-algebra $\mathcal{Y} = \{\emptyset, Y\}$ since the pre-images of these elements are always $\emptyset$ and $X$. In the following, we will propose a computable and monotone $pre^\sharp_f$ that is based on $img^\sharp_f$ and a finite partition of input $X$.

**Lemma 5.18.** *Let $f \colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$ be a measurable function, let the function $img^\sharp_f \colon \wp(X) \to \wp(Y)$ over-approximate $img_f$, and let $\mathbf{T}$ denote the set of all partitions over $X$. We define a function $pre^\sharp_f \colon \mathbf{T} \to (\wp(Y) \to \wp(X))$ as*

$$pre^\sharp_f[T](B) \triangleq \bigcup \{t \in T \mid img^\sharp_f(t) \cap B \neq \emptyset\}$$

*Then,*

$$pre_f \preceq pre^\sharp_f[T]$$

*Proof.*

$$\begin{aligned} pre_f(B) &= \{x \in X \mid img_f(\{x\}) \cap B \neq \emptyset\} \\ &\subseteq \{x \in X \mid t \in T \wedge x \in t \wedge img_f(\{t\}) \cap B \neq \emptyset\} \\ &\subseteq \{x \in X \mid t \in T \wedge x \in t \wedge img^\sharp_f(\{t\}) \cap B \neq \emptyset\} \\ &\subseteq \bigcup \{t \in T \mid img^\sharp_f(\{t\}) \cap B \neq \emptyset\} \\ &= pre^\sharp_f[T](B) \end{aligned}$$

**Proposition 5.19.** *$pre^\sharp_f[T]$ is closed under union,* i.e. $pre^\sharp_f[T](\bigcup_{A \in \mathbf{A}} A) = \bigcup_{A \in \mathbf{A}} pre^\sharp_f[T](A)$.

*Proof.*

$$\begin{aligned} & a \in pre^\sharp_f[T](A \cup B) \\ \Leftrightarrow\ & \exists t \in T : a \in t \wedge img^\sharp_f(t) \cap (A \cup B) \neq \emptyset \\ \Leftrightarrow\ & \exists t \in T : a \in t \wedge (img^\sharp_f(t) \cap A \neq \emptyset) \ \vee \ (img^\sharp_f(t) \cap B \neq \emptyset) \\ \Leftrightarrow\ & \exists t \in T : (a \in t \wedge img^\sharp_f(t) \cap A \neq \emptyset) \ \vee \ (a \in t \wedge img^\sharp_f(t) \cap B \neq \emptyset) \\ \Leftrightarrow\ & a \in pre^\sharp_f[T](A) \cup pre^\sharp_f[T](B) \end{aligned}$$

**Corollary 5.20.** *For any partition $T$ over $X$, $pre^\sharp_f[T]$ is monotone.*

We can now apply Theorem 5.10 to this computable $pre^\sharp[T]$ and obtain computable upper and lower probability bounds for the output events.

**Theorem 5.21.** *Let $(X, \mathcal{X}, \mu)$ be a probability space, and let $f\colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$ be a measurable function that induces the output probability measure $\mu_f\colon \mathcal{Y} \to [0,1]$,* i.e. *$\mu_f = \mu \circ pre_f$.*

*Given a function $img^\sharp_f\colon \wp(X) \to \wp(Y)$ such that $img_f \preceq img^\sharp_f$, a finite partition $T$ over $X$, and a monotone abstraction $\uparrow\colon \wp(X) \to \mathcal{X}$, we let*

$$pre'^\sharp_f[T](A) \triangleq \uparrow\big(\bigcup\{t \mid \exists t \in T\colon img^\sharp_f(t) \cap A \neq \emptyset\}\big) \text{ and}$$
$$pre'^\flat[T]_f(A) \triangleq pre'^\sharp_f[T](A^{\complement})^{\complement}$$

*and we define $\mu^\sharp_f \triangleq \mu \circ pre'^\sharp_f[T]$ and $\mu^\flat_f \triangleq \mu \circ pre'^\flat_f[T]$. Then,*

*(i)* $\mu^\flat_f(A) \leq \mu_f(A) \leq \mu^\sharp_f(A)$
*(ii)* $\mu^\flat_f(A) = 1 - \mu^\sharp_f(A^{\complement})$, *and*
*(iii) $\mu^\flat_f$ and $\mu^\sharp_f$ are monotone.*

*Proof.* The function $pre^\sharp_f[T]$ over-approximates $pre_f$ (by Lemma 5.18), and it is monotone (by Proposition 5.20). Thus, the above is a direct consequence of Theorem 5.10.

### 5.3.1 Choice of partition

Recall that we assumed that we knew the input probability measure, and thus we know the input space $(X, \mathcal{X})$. This provides us a better basis for choosing a good partition. When we choose the partition $T$ such that the elements $t \in T$ are measurable in the input space $\mathcal{X}$, *i.e.* $t \in \mathcal{X}$, we can simplify the expressions for the upper and lower probability bounds. When the elements are measurable, $pre^\sharp_f[T](A) \in \mathcal{X}$ for every output event $A$. This reduces the abstraction $\uparrow$ to the identity function, and we can unfold the $\mu^\sharp_f$ and $\mu^\flat_f$ into a simpler form; see Theorem 5.23.

**Lemma 5.22.** *If $T$ is a partition whereby $T \subseteq \mathcal{X}$, then $pre^\sharp_f[T](A) \in \mathcal{X}$.*

*Proof.* Let $B \in \wp(Y)$, and let $T \subseteq \mathcal{X}$ by any finite partition over $X$. By the definition of $T$, $t \in T \Rightarrow t \in \mathcal{X}$. Since $T$ is finite and $\mathcal{X}$ is closed under countable union, $A \in \wp(T) \to A \in \mathcal{X}$. The set $\bigcup\{t \in T \mid img^\sharp_f(\{t\}) \cap B \neq \emptyset\} \in \wp(T)$, and thereby, it also exists in $\mathcal{X}$.

**Theorem 5.23.** *Let $f\colon (X, \mathcal{X}) \to (Y, \mathcal{Y})$ be a measurable function with the image function $img_f$ and the pre-image function $pre_f$, let $(X, \mathcal{X}, \mu)$ be a probability space, let $img^\sharp_f\colon \wp(X) \to \mathcal{Y}$ be a function that over-approximates $img_f$, and let $T$ be a finite partition over $X$ such that $T \subseteq \mathcal{X}$. Then,*

$$\mu^\sharp_f(A) = \sum_{t\in T, img^\sharp(t)\cap A\neq\emptyset} \mu(t)$$
$$\mu^\flat_f(A) = \sum_{t\in T, img^\sharp(t)\subseteq A} \mu(t)$$

*Proof.* Let $T_A = \{t \in T \mid img^\sharp_f(t) \cap A \neq \emptyset\}$, whose elements are pair-wise disjoint since $T_a \subseteq T$ and $T$ is a partition. In addition, let $\uparrow$ be the identity function *id*.

$$\mu^\sharp_f(A) = \mu(id(pre^\sharp_f[T](A))) = \mu(\bigcup\{t \in T \mid img^\sharp_f(t) \cap A \neq \emptyset\}) = \mu(\bigcup_{t\in T_A} t)$$
$$= \sum_{t\in T_A} \mu(t) = \sum_{t\in T, img^\sharp_f(t)\cap A\neq\emptyset} \mu(t)$$

For the under-approximating part, we first show that $pre^\flat_f[T](A) = \{t \in T \mid img^\sharp_f(t) \subseteq A\}$.

$$\begin{aligned}
pre^\flat_f[T](A) &= id(pre^\sharp_f[T](A)) \setminus id(pre^\sharp_f[T](A^\complement)) \\
&= \bigcup\{t \in T \mid img^\sharp_f(t) \cap A \neq \emptyset\} \setminus \bigcup\{t \in T \mid img^\sharp_f(t) \cap A^\complement \neq \emptyset\} \\
&= \bigcup\{t \in T \mid img^\sharp_f(t) \cap A \neq \emptyset \wedge \neg \left(img^\sharp_f(t) \cap A^\complement \neq \emptyset\right)\} \\
&= \bigcup\{t \in T \mid img^\sharp_f(t) \cap A \neq \emptyset \wedge \left(img^\sharp_f(t) \cap A^\complement = \emptyset\right)\} \\
&= \bigcup\{t \in T \mid img^\sharp_f(t) \cap A \neq \emptyset \wedge \left(img^\sharp_f(t) \subseteq A\right)\} \\
&= \bigcup\{t \in T \mid img^\sharp_f(t) \subseteq A\}
\end{aligned}$$

Now, we may use a similar argument as in the over-approximating case, where $T_A = \{t \in T \mid img^\sharp_f(t) \subseteq A\}$ (the elements are pair-wise disjoint, and $T_A$ is a subset of the partition $T$, whose elements are pair-wise disjoint).

$$\mu(pre^\sharp_f[T](A)) = \mu(\bigcup\{t \in T \mid img^\sharp_f(t) \subseteq A\}) = \mu(\bigcup_{t\in T_A} t)$$
$$= \sum_{t\in T_A} \mu(t) = \sum_{t\in T, img^\sharp_f(t)\subseteq A} \mu(t)$$

This concludes the proof.

### 5.3.2 Tightest probability bounds

When $img^\sharp$ is monotone, the choice of $T$ influences the tightness of the probability bounds. Therefore, we provide some observations on that choice.

**Definition 5.24.** *A partition $T$ over $X$ is* finer *than $T'$ if every element of $T$ is a subset of an element in $T'$. The* singleton partition *$T$ over $X$ consists of the singletons $T = \{\{x\} \mid x \in X\}$.*

**Corollary 5.25.** *The finest partition $T$ over $X$ is the singleton partition.*

**Lemma 5.26.** *Let $T$ and $T'$ be two partitions over $X$, where $T$ is finer than $T'$. If $img_f^\sharp$ is monotone, then $pre_f^\sharp[T] \preceq pre_f^\sharp[T']$.*

*Proof.* We show that $a \in pre_f^\sharp[T](A) \Rightarrow a \in pre_f^\sharp[T'](A)$: The assumption $a \in pre_f^\sharp[T](A)$ implies that there is $t \in T$ such that $a \in t \wedge img_f^\sharp(t) \cap A \neq \emptyset$. Since $T$ is finer than $T'$, for any $t \in T$, there exist a $t' \in T'$ such that $t \subseteq t'$, that is, $a \in t \Rightarrow a \in t'$. Furthermore, due to monotonicity of $img_f^\sharp$, $t \subset t' \Rightarrow img_f^\sharp(t) \subseteq img_f^\sharp(t')$. We now recall that if $B \subseteq B'$, then $B \cap A \neq \emptyset \Rightarrow B' \cap A \neq \emptyset$, which applies to the over-approximated images $img_f^\sharp(t) \subseteq img_f^\sharp(t')$ $img_f^\sharp(t) \cap A \neq \emptyset \Rightarrow img_f^\sharp(t') \cap A \neq \emptyset$. Hence, we find that $\exists t \in T\colon a \in t \wedge img_f^\sharp(t) \cap A \neq \emptyset \Rightarrow \exists t' \in T'\colon a \in t' \wedge img_f^\sharp(') \cap A \neq \emptyset$. Thus, $a \in pre_f^\sharp[T'](A)$.

**Corollary 5.27.** *Let $img_f^\sharp$ be monotone. If $T$ is the singleton partition over $X$, then $pre_f^\sharp[T]$ is the minimal over-approximation of $pre_f$, that is, $\forall T' \in \mathbf{T}\colon pre_f^\sharp[T] \preceq pre_f^\sharp[T']$.*

*Proof.* A consequence of Corollary 5.25 and Lemma 5.26.

**Lemma 5.28.** *For a countable infinite $X$, there is no finest finite partition $T$.*

*Proof.* By contradiction: Assume that we have a finest finite partition $T$. Since $X$ has $\aleph_0$ elements and $T$ has $n \in \mathbb{N}$ elements, there must exist a $t \in T$ such that $|t| \geq 2$; thus, $x \in t$ such that $t \setminus \{x\} \neq \emptyset$. Thus, the partition $T' \triangleq (T \setminus t) \uplus (t \setminus \{x\}) \uplus \{x\}$ is also a partition, and it is a finer partition than $T$. Thus, $T$ is not the finest partition.

In some cases, when the probability space $(X, \mathcal{X}, \mu)$ consists of a finite $\sigma$-algebra, it is not important to have a finest partition $T$; if $\mathcal{X}$ is a finite $\sigma$-algebra, then there exists a finite partition $T$ over $X$ such that $\sigma(T) = \mathcal{X}$ [69]. In these cases, the partition $T$ is measurable. Thus, if we choose a partition $T'$ finer than $T$, then the best abstraction $\uparrow$ abstracts $pre[T'](A)$ to $pre[T](A)$; if we choose an incomparable partition $T''$, then $\uparrow pre[T''](A) \supseteq pre[T](A)$ since the $pre[T''](A)$ will be abstracted to the set of $T$-partition elements with which it overlaps.

## 5.4 Combining analyses

In this section we assume that we have two forwards or backwards analyses and want to combine them into a stronger result. For the case where we have two pre-image over-approximating functions $pre_f^\sharp, pre'^\sharp_f\colon \wp(Y) \to \wp(X)$ we show how to combine them to a tighter pre-image over-approximating function $pre''^\sharp_f\colon \wp(Y) \to \wp(X)$. For the cases where we have one pre-image over-approximating function and one image over-approximating function, or two image over-approximating functions, we can apply the same method after creating the pre-image over-approximating function(s).

However, in case of two image over-approximating functions it may be more convenient to combine them directly -we shall see an example of this in Section 5.5.3 page 78.

First, we combine two backwards analysis to a sometimes stronger result.

**Lemma 5.29.** *Let* $pre_f, pre^\sharp_f, pre'^\sharp_f \colon \wp(Y) \to \wp(X)$ *be three functions such that* $pre_f \preceq pre^\sharp_f$ *and* $pre_f \preceq pre'^\sharp_f$. *Then*

$$pre_f(A) \subseteq pre^\sharp_f(A) \cap pre'^\sharp_f(A).$$

*Proof.* Let $x \in pre_f(A)$ then $x \in pre^\sharp_f(A)$ and $x \in pre'^\sharp_f(A)$ by assumption. Thus, $x \in pre^\sharp_f(A) \cap pre'^\sharp_f(A)$.

**Proposition 5.30.** *If* $pre^\sharp_f(A) \in \mathcal{X}$ *and* $pre'^\sharp_f(A) \in \mathcal{X}$ *whenever* $A \in \mathcal{Y}$, *then* $(pre^\sharp_f(A) \cap pre'^\sharp_f(A)) \in \mathcal{X}$ *whenever* $A \in \mathcal{Y}$.

*Proof.* By Proposition 2.5, $\mathcal{X}$ is closed under countable intersections and specifically the intersection of any two elements in $\mathcal{X}$.

Equivalently, if we have two image-over-approximating functions, the following lemma allows us to combine them into a perhaps stronger result. The lemma is related to Lemma 5.29.

**Lemma 5.31.** *Let* $img_f, img^\sharp_f, img'^\sharp_f \colon \wp(X) \to \wp(Y)$ *be three functions such that* $img_f \preceq img^\sharp_f$ *and* $img_f \preceq img'^\sharp_f$. *Then,*

$$img_f(A) \subseteq img^\sharp_f(A) \cap img'^\sharp_f(A).$$

*Proof.* Let $x \in img_f(A)$; then, $x \in img^\sharp_f(A)$ and $x \in img'^\sharp_f(A)$ by assumption. Thus, $x \in img^\sharp_f(A) \cap img'^\sharp_f(A)$.

## 5.5 Case studies

In this section, we describe three experimental results for forward analysis; the analysis used will be sign, interval and termination analysis. In the first two cases, we assume that the analysed programs are terminating, but in the last case, we have chosen a non-terminating program. When a program output may yield a non-terminating computation, it influences the lower bounds, and we show how to combine a sign analysis and a termination analysis to achieve tight lower bounds (and probability bounds for non-termination).

### 5.5.1 A sign analysis

In this first case, we analyse a terminating program `sqr` in Figure 5.1a that computes the square of `x` and returns the result via the variable `y`. The program `sqr` has the input-output function $|\texttt{sqr}|\colon \mathbb{Z} \to \mathbb{Z}$

$$|\texttt{sqr}|(n) = n \cdot n.$$

In the following, we use $\mathbb{Z}^+$ and $\mathbb{Z}^-$ as abbreviations for all the positive and negative numbers, respectively, in $\mathbb{Z}$. We assume the input probability space $(\mathbb{Z}, \mathcal{Z}, \mu)$ as defined in Figure 5.1b.

```
int sqr(int x){
 int y;
 y := x;
 y := y*x;
 return y;
}
```

(a)

| $t \in T$ | $\mu(t)$ |
|---|---|
| $\mathbb{Z}^-$ | $1/3$ |
| $\{0\}$ | $1/4$ |
| $\mathbb{Z}^+$ | $5/12$ |

(b)

| $t \in T$ | $img^\sharp_{\lvert\texttt{sqr}\rvert,\text{SIGN}}$ |
|---|---|
| $\mathbb{Z}^-$ | $\mathbb{Z}^+$ |
| $\{0\}$ | $\{0\}$ |
| $\mathbb{Z}^+$ | $\mathbb{Z}^+$ |
| $\mathbb{Z}^- \cup \{0\}$ | $\mathbb{Z}^+ \cup \{0\}$ |
| $\mathbb{Z}^- \cup \mathbb{Z}^+$ | $\mathbb{Z}^- \cup \mathbb{Z}^+$ |
| $\mathbb{Z}^+ \cup \{0\}$ | $\mathbb{Z}^+ \cup \{0\}$ |
| $\mathbb{Z}$ | $\mathbb{Z}$ |

(c)

Fig. 5.1: The analysed program (a), the input distribution (b) and the sign-analysis output (c).

We will demonstrate the consequences of the partition choice, and thus, we analyse `sqr` using three different partitions $T_1 = \{\mathbb{Z}^-, \{0\}, \mathbb{Z}^+\}$ $T_2 = \{\{0\}, \mathbb{Z}^+ \cup \mathbb{Z}^-\}$, and $T_3 = \{\mathbb{Z}^- \cup \{0\}, \mathbb{Z}^+\}$.

We will use a black-box sign analysis that provides a monotone over-approximation of the program's image function $img^\sharp_{|\texttt{sqr}|,\text{SIGN}}\colon \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$; see Figure 5.1c. More specifically, we will use a sign analysis that is an extension of that described in chapter 7 in [114]. We let the measurable space of the output be $\wp(\{\mathbb{Z}^-, \{0\}, \mathbb{Z}^+\})$. This analysis assumes the independence of the variables and provides a monotone function from the power set over inputs to the power set over outputs.

In this sign analysis, a program is seen as a transition system over program points whereby a state, *i.e.* a mapping from the program variables $\mathbf{Var}$ to the values $\mathbb{Z}$, is transformed between program points. The program's input and output are the states at the program's initial and final program points, respectively. The sign analysis transforms abstract states, *i.e.* a mapping from the program variable $\mathbf{Var}$ to a set of possible signs $\mathbf{Sign} = \wp(\{\text{NEG}, \text{ZERO}, \text{POS}\})$, that is, $\mathbf{Var} \to \mathbf{Sign}$. The analysis relies on an abstraction function from values to signs $\alpha_{\mathbf{Sign}}\colon \wp(\mathbb{Z}) \to \wp(\mathbf{Sign})$ and a concretization function from signs to values $\gamma_{\mathbf{Sign}}\colon \wp(\mathbf{Sign}) \to \wp(\mathbb{Z})$.

$$\alpha_{\mathbf{Sign}}(A) = \{\alpha'_{\mathbf{Sign}}(a) \mid a \in A\} \quad \text{where } \alpha'_{\mathbf{Sign}}(n) = \begin{cases} \text{POS} & n > 0 \\ \text{ZERO} & n = 0 \\ \text{NEG} & n < 0 \end{cases}$$

$$\gamma_{\mathbf{Sign}}(S) = \bigcup\{\gamma'_{\mathbf{Sign}}(s) \mid s \in S\} \text{ where } \gamma'_{\mathbf{Sign}}(s) = \begin{cases} \mathbb{Z}^+ & s = \text{POS} \\ \{0\} & s = \text{ZERO} \\ \mathbb{Z}^- & s = \text{NEG} \end{cases}$$

Instead of using the entire states as output, we extract the mapping for the specified output variable $\mathtt{y}$ at the final program point. The pre-image functions can be indicated by their result on each of $\mathbb{Z}^-$, $\{0\}$, and $\mathbb{Z}^+$.

$$\begin{array}{l} pre_{|\mathtt{sqr}|}[T_1](\mathbb{Z}^-) = \mathbb{Z}^+, \; pre_{|\mathtt{sqr}|}[T_2](\mathbb{Z}^-) = \mathbb{Z}^- \cup \mathbb{Z}^+, \\ pre_{|\mathtt{sqr}|}[T_1](\{0\}) = \{0\}, \; pre_{|\mathtt{sqr}|}[T_2](\{0\}) = \{0\}, \\ pre_{|\mathtt{sqr}|}[T_1](\mathbb{Z}^+) = \mathbb{Z}^+, \; pre_{|\mathtt{sqr}|}[T_2](\mathbb{Z}^+) = \mathbb{Z}^- \cup \mathbb{Z}^+, \end{array} \quad \begin{array}{l} pre_{|\mathtt{sqr}|}[T_3](\mathbb{Z}^-) = \emptyset, \\ pre_{|\mathtt{sqr}|}[T_3](\{0\}) = \mathbb{Z}^- \cup \{0\}, \\ pre_{|\mathtt{sqr}|}[T_3](\mathbb{Z}^+) = \mathbb{Z}^+. \end{array}$$

If we calculate the results of the output events $\mathbb{Z}^- \cup \{0\}$ and $(\mathbb{Z}^- \cup \{0\})^{\complement} = \mathbb{Z}^+$), they differ.

$$\begin{array}{ll} pre_{|\mathtt{sqr}|}[T_1](\mathbb{Z}^+) = \mathbb{Z}^+ & pre_{|\mathtt{sqr}|}[T_1](\mathbb{Z}^- \cup \{0\}) = \mathbb{Z}^+ \cup \{0\} \\ pre_{|\mathtt{sqr}|}[T_2](\mathbb{Z}^+) = \mathbb{Z}^- \cup \mathbb{Z}^+ & pre_{|\mathtt{sqr}|}[T_2](\mathbb{Z}^- \cup \{0\}) = \mathbb{Z} \\ pre_{|\mathtt{sqr}|}[T_3](\mathbb{Z}^+) = \mathbb{Z}^+ & pre_{|\mathtt{sqr}|}[T_3](\mathbb{Z}^- \cup \{0\}) = \mathbb{Z}^- \cup \{0\} \end{array}$$

These results imply that the upper and lower probability bounds that they define differ on these two events; this affects both the upper and lower bounds since the lower probability bound was defined using the dual of the over-approximating pre-images, *i.e.*, those of the complement set. More precisely, they define the upper and lower probability bounds of the output events shown in Figure 5.2. The bounds created using the partition $T_1$ is not surprisingly the most accurate; however, the fact that it is as precise as the correct output probability measure is due to two facts: (i) the input space $\mathcal{Z} = \sigma(T_1)$ and (ii) the analysis that we used were precise for all partition elements in $T_1$.

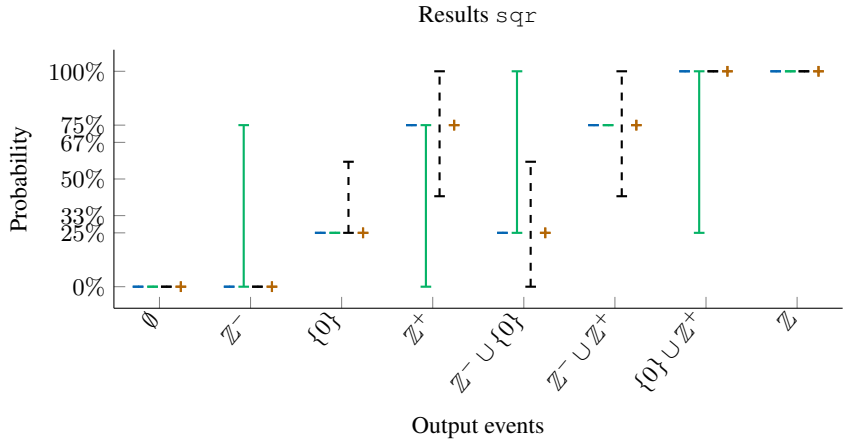



Fig. 5.2: The upper and lower probability bounds for $pre_f^\sharp[T_1]$ (blue), $pre_f^\sharp[T_2]$ (green solid), $pre_f^\sharp[T_3]$ (black dashed), together with the correct output probability (orange "+") for the measurable output event.

### 5.5.2 Interval Analysis

For these programs, we will not present the pre-image functions; rather, we only provide the resulting upper and lower bounds, *e.g.* the image function of the program `f1` involves 10000 input combinations for the first result alone.

We analyse three terminating programs using the interval analysis, `f2`, `g2`, and `h`, presented in Figure 5.3. The bodies of the programs `f1` in Figure 5.3 [95] and `g1` in Figure 5.3[3] were analysed by Monniaux's experimental analysis. The last program `h` in Figure 5.3 has been analysed with our reproduction of Monniaux's experimental analysis. Monniaux's analysis handles both probabilistic programs, *i.e.* programs with random generators, and deterministic programs, whereas our presented technique only handles deterministic programs. His analysis unfolds the loops; therefore, we will also unfold the loops and transform the test programs `f1` and `g1` into deterministic programs `f2` and `g2`, where the results of the random generator calls are given as input.

For the program `f1`, Monniaux produced three probability measures over outputs, where he experimented with what corresponds to finer and coarser partitions and output $\sigma$-algebras. Since the input variables, *i.e.* the random generators, are independent and since the measurable input space is the product of their spaces, the measurable space can be uniquely defined by the measurable spaces for each variable. We let our input partition correspond to the Monniaux's parametrized abstraction for the results to be comparable. All probability measures for the input variables are uniform distributions over $[0, 1]$, and they are partitioned by a parameter $N$ into equally

[3] personal communication with D. Monniaux.

```
double f1() {
  double x=0.0;
  int i;

  for (i=0; i<4; i++)
    x += drand48()*2.0-1.0;
  return x; }
```

```
double g1() {
  double x=0.0;
  int i;
  if(drand48()>0.5)
    x += drand48()*2.0-1.0;
  for (i=0; i<3; i++)
    x += drand48()*2.0-1.0;
  return x;}
```

```
double f2 (double x1,
  double x2, double x3,
  double x4) {
  double x;
  x = 0.0;
  x = x+ x1*2.0-1.0;
  x = x+ x2*2.0-1.0;
  x = x+ x3*2.0-1.0;
  x = x+ x4*2.0-1.0;
  return x;}
```

```
double g2(double x1,
  double x2, double x3,
  double x4, double x5) {
  double x;
  x = 0.0;
  if (x5 >= 0.5)
    x = x+ x1*2.0-1.0;
  x = x+ x2*2.0-1.0;
  x = x+ x3*2.0-1.0;
  x = x+ x4*2.0-1.0;
  return x;}
```

```
int h(double x, int a) {
  if(a<3){x=x+2;} else{x=x+3;}
  if(a>=3){x=x+2;} else{x=x+3;}
  if(a<2){x=x-10;} else{x=x;}
  if(a>6||a<2){x=x+5;}else{x=x-5;}
  return x;
}
```

Fig. 5.3: The test programs `f1`, `f2`, `g1`, `g2`, and `h`; the programs `f1` and `f2` are equivalent and `g1` and `g2` are equivalent.

sized parts, *e.g.*, $N = 2$; then, the partition $T = \{[0, 0.5], [0.5, 1.0]\}$. The input space is then created by the Cartesian product of the variables, *e.g.*, $T \times T \times T \times T$.

**Results.** In the following, Monniaux's result will be presented in blue (to the left), our results are presented in green (to the right), and the correct probability for the interval is indicated by a solid black line. Each result will be titled with the program name, the parameter $N$ and the length of the input intervals; the output intervals are shown using vertical white lines. The output intervals are indicated by white vertical lines in the figures.

*Program `f`.* Figure 5.4 shows Monniaux's upper bound and our upper bound results for the program `f` are equivalent. One advantage is that our approach also provides useful lower bounds; note that we used the interval analysis to provide an image-function from reals to reals (since the program was terminating). Had we not used the knowledge that the program terminated the lower bounds would all have been 0.

*Program `g`.* The results for program `g` (Figure 5.5) reveal some differences in the upper bounds between Monniaux's experimental analysis and the presented technique;

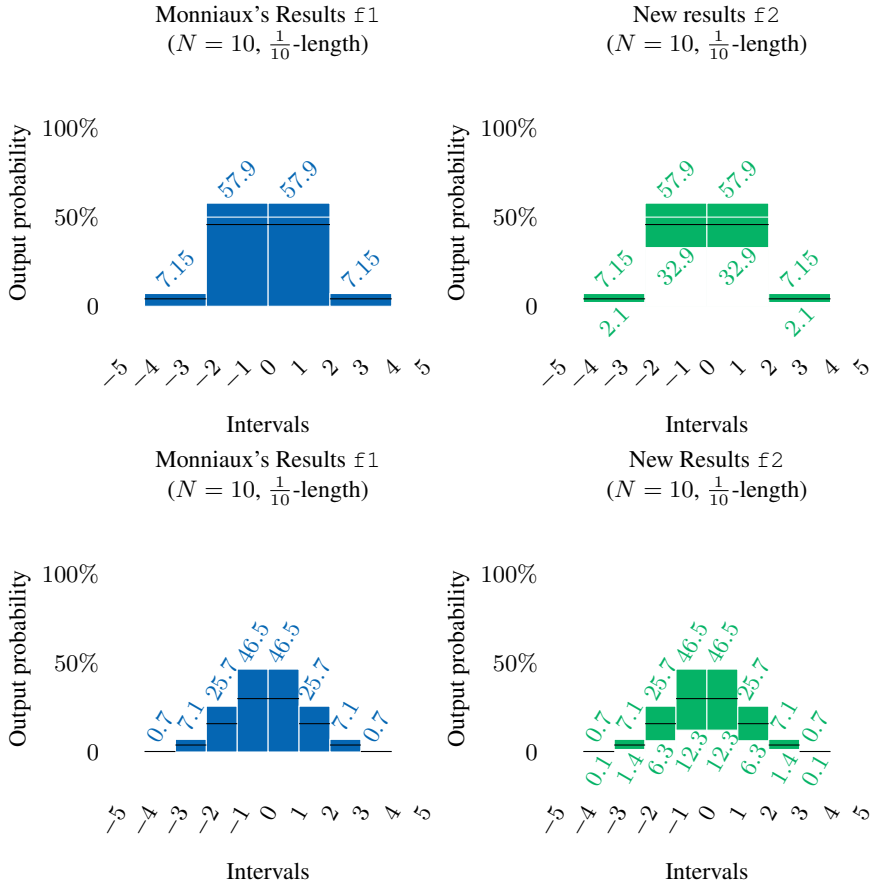



Fig. 5.4: The test results for program `f1` and the similar `f2` with equivalent input abstractions and partitions, those of length 1/10; above, the output events are intervals of length 2, and below the output events are intervals of length 1.

the lower bounds provided by our approach are all 0. In the first example, we have used the partition $\{[0, 0.33], [0.33, 0.67], [0.67, 1.0]\}$ for all inputs (they are independent). Here, we obtain the upper probability bound $83\%$ for the output $[0, 1]$ (compared to Monniaux's $108\%$ or, $100\%$ when reducing to the obvious maximum upper bound.). One question concerns where the imprecision stems from; to answer this, we changed the partition of the variable `x5` so that the condition in the if-expression could be determined to be only true or only false. This reduced the upper bounds 1-2% *e.g.*, from $83\%$ to $81\%$ for interval $[0, 1]$. However, this is still imprecise when comparing to the correct result shown as the solid black lines, *e.g.*, $32\%$ for interval $[0, 1]$. From this observation, we may conclude that the abstract domain/intput partition is too imprecise and a refinement is needed. As expected after the discussion on choosing the "best" partition (and as Monniaux noted when evaluating his results)

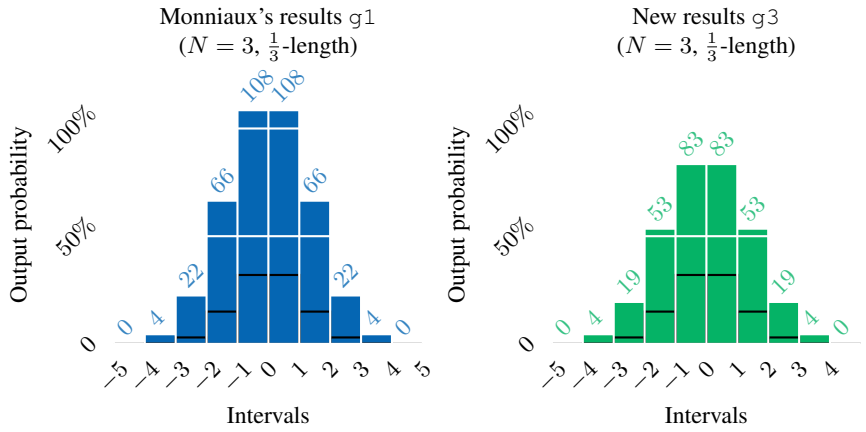



Fig. 5.5: The test results for program `g1` and the similar `g2` with equivalent input abstractions and partitions, those of length 1/3. The output events are intervals of length 1.

when we choose a finer input partition/abstract domain, we obtain more precise results. We saw this in the results for `f`.

A natural question is when can we use coarse analyses? One answer is that it provides guarantees such that "at least $95\%$ of the output lies in $[-3, 3]$ regardless of the probability distribution of `x5`".

*Program* `h`. The difference between the results for `g` arise from the treatment of the if-expression `if b then e1 else e2`. Monniaux's experimental analysis [95] is a forward analysis, and for each input $t$ (corresponding to the partition elements), he computes the possible output and the associated probabilities. For an if-expression, Monniaux's correct analysis computes whether `b` may be true or may be false and finds two sets of outputs: the possible output $O_{\texttt{e1}}$ of `e1` and the possible output $O_{\texttt{e2}}$ of `e2`. If `b` may be true, the analysis assigns the input probability of $t$, namely, $\mu(t)$, to $O_{\texttt{e1}}$, and if `b` may be false, the analysis also assigns $\mu(t)$ to $O_{\texttt{e2}}$. The upper probability bound of an output event $A$ is the sum of "probabilities" of the outputs that overlap with $A$. Thus, when $A$ overlaps with both $O_{\texttt{e1}}$ and $O_{\texttt{e2}}$, Monniaux's analysis calculates $\mu(t) + \mu(t)$ as the upper bound, whereas $\mu(t)$ suffices (and is the upper bound calculated by the technique presented in this paper). In summary, the probability that an output $A$ is the probability of each input $p$ with the feasible path with output in $A$ multiplied by the number $n$ of these feasible paths, *i.e.*, $p \cdot n$.

To show how this affects the results, we have analysed program `h` for the probabilities of the output events $[0, 1]$ and $[10, 11]$. We assume an input measure such that `x` $\in [0, 1]$ with probability 1, and there is a $33.33\%$ chance of `a` $\in \{0, 1, \ldots, 6\}$ and an $66.67\%$ chance of `a` $\in \{7, 8, 9, 10\}$. Thus, the correct probability of the output $[0, 1]$ is $33.33\%$ and that of $[10, 11]$ is $66.67\%$. In program `h` there are several feasible execution paths from each partition element to an output; each path adds either 0 to the input variable `x` or 10 to it. The following table indicates the feasible paths and their output, *e.g.*, when `a` is 1, the first condition (`a<3`) is true, the second (`a>=3`) is

false, the third (`a<2`) is true, and the fourth (`(a>6||a<2)`) is true.

| value of `a` | `a<3` | `a>=3` | `a<2` | `(a>6\|\|a<2)` | output |
|---|---|---|---|---|---|
| 1 | T | F | T | T | $[0,1]$ |
| 2 | T | F | F | F | $[0,1]$ |
| 3,4,5,6 | F | T | F | F | $[0,1]$ |
| 7,8,9,10 | F | T | F | T | $[10,11]$ |

Assuming that the underlying analysis is able to exclude infeasible paths, Monniaux's analysis will find 3 feasible paths for $\texttt{a} \in \{0,1,\ldots,6\}$ that all lead to $[0,1]$ and one feasible path for $\texttt{a} \in \{7\ldots,10\}$ that leads to $[10,11]$. We see, in Figure 5.6, that Monniaux's approach to if-expressions results in a large over-approximation of the output $[0,1]$, namely, $100\%$, and the our new results are similar to the correct probabilities.

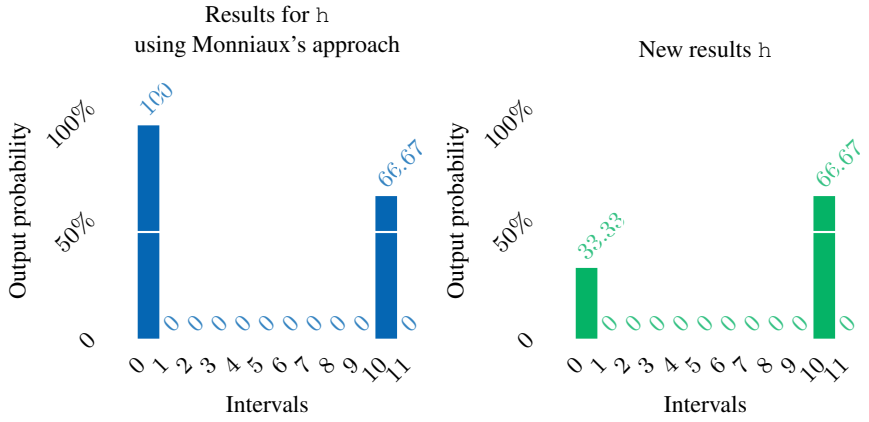



Fig. 5.6: The results for program `h`.

### 5.5.3 Nontermination and intersection of two analyses

In the other cases, the analysed programs were all terminating, and that ensured that the input-output relation was a total function from input to output. However, not all programs are terminating, that is, each program input yields either a non-terminating computation or an output result. Therefore, we extend the outputs $Y$ with a special character $\perp$, which indicates non-termination, *i.e.* $Y_\perp = Y \uplus \perp$,

Analyses such as sign and interval analyses are intended to obtain partial correctness; they yield that *if the program terminates, the analysis' output contains the concrete program result*, see *e.g.* [114]. Such analyses do (obviously) not conclude anything about termination/non-termination, and we simply assume that the

concretization of the analysis output also yields a non-terminating computation. For instance, for the sign analysis, we use the following abstraction and concretization functions for the input/output variable values.

$$\begin{aligned} \alpha\colon \wp(\mathbb{Z}_\perp) &\to \wp(\mathbf{Sign}) & \alpha(A) &\triangleq \alpha_{\mathbf{Sign}}(A \cap \mathbb{Z}) \\ \gamma\colon \wp(\mathbf{Sign}) &\to \wp(\mathbb{Z}_\perp) & \gamma(S) &\triangleq \gamma_{\mathbf{Sign}}(S) \cup \{\perp\} \end{aligned}$$

For example, if y is the output variable and the sign analysis yields$\{\mathtt{y} \to \{\text{ZERO}\}\}$, we concretize the sign to $\gamma(\{\text{ZERO}\}) = \{0, \perp\}$. Using this more precise concretization decreases the lower probability bounds for the input yielding terminating computations. For instance, the lower probability bound of $\{0\}$ of any program becomes $0$ because there is no guarantee that $A$ (or any other input) will result in $\{0\}$; all abstract states concretize to $\{\perp\}$ and something more that is not a subset of $\{0\}$.

```
int (sum)(int x){
    int y = 0;
    while (x!= 0){
        y = y + x;
        x = x - 1;
    }
    return y;
}
```

(a)

| $t \in T$ | $\mu(t)$ |
|---|---|
| $\mathbb{Z}^-$ | $1/3$ |
| $\{0\}$ | $1/4$ |
| $\mathbb{Z}^+$ | $5/12$ |

(b)

| $t \in T$ | $img^\sharp_{\lvert\mathtt{sum}\rvert,\text{SIGN}}$ |
|---|---|
| $\mathbb{Z}^-$ | $\{\perp\} \cup \mathbb{Z}^-$ |
| $\{0\}$ | $\{\perp\} \cup \{0\}$ |
| $\mathbb{Z}^+$ | $\{\perp\} \cup \{0\} \cup \mathbb{Z}^+$ |

(c)

Fig. 5.7: The analysed program (a), the input distribution (b) and the sign-analysis output (c).

#### 5.5.3.1 Sign analysis

In this case, we analyse a non-terminating program sum in Figure 5.7a that sums the values from x to zero and yields a non-terminating computation when fed a negative value. The program sum has the input-output function $|\mathtt{prg}|\colon \mathbb{Z} \to \mathbb{Z}_\perp$.

$$|\mathtt{sum}|(n) = \begin{cases} \perp & n < 0 \\ \sum_{i=0}^{n} i & n \leq 0 \end{cases}$$

We analyse sum with respect to an input probability space $(\mathbb{Z}, \mathcal{Z}, \mu)$, as defined in Figure 5.7b, using the partition $T = \{\mathbb{Z}^-, \{0\}, \mathbb{Z}^+\}$; we apply the sign analysis and obtain $img^\sharp_{|\mathtt{sum}|}\colon \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$, as provided in Figure 5.7c. We let the measurable space of the output be $(\mathbb{Z}_\perp, \wp(\{\perp, 0, \mathbb{Z}^-, \mathbb{Z}^+\}))$.

From Theorem 5.23, the probability of each partition element $t$ together with the analysis output $img^\sharp_{|\mathtt{sum}|}(t)$ defines the upper and lower probability bounds for every

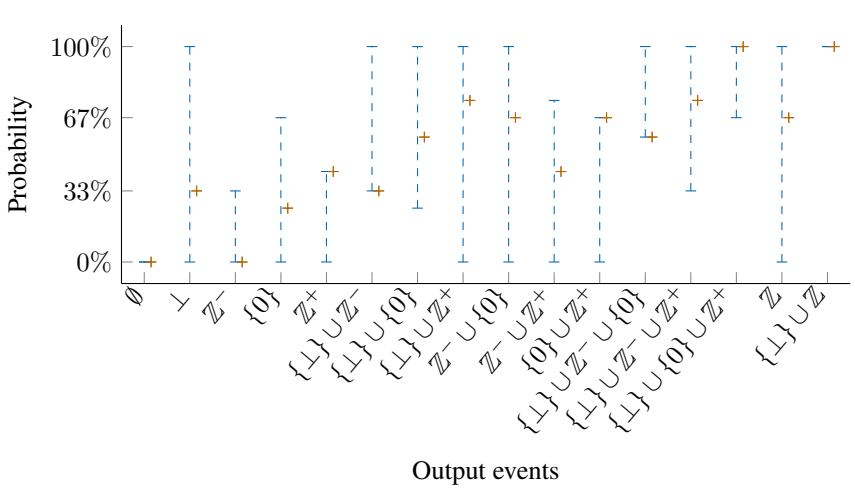
output (including non-termination). We have provided these results in Figure 5.7. The upper and lower probability bounds for non-termination are 1 and 0, respectively, which is correct but not informative.

The upper and lower probability bounds for $\{0\}$ are as follows.

$$\mu(pre_f^\sharp[T](\{0\})) = \sum_{t \in T, img^\sharp(t) \cap A \neq \emptyset} \mu(t) = \sum_{t \in \{\{0\}, \mathbb{Z}^+\}} \mu(t) = \frac{1}{4} + \frac{5}{12} = \frac{2}{3}$$

$$\mu(pre_f^\flat[T](\{0\})) = \sum_{t \in T, img^\sharp(t) \subseteq A} \mu(t) = \sum_{t \in \emptyset} \mu(t) = 0$$

If we had avoided the possible non-termination, then $img^\sharp(\{0\}) \subseteq \{0\}$, and the lower probability bound would have been tighter.




#### 5.5.3.2 Termination analysis

We suggest applying a termination analysis to variations of the program obtaining $img^\sharp_{|\texttt{sum}|,\text{TERM}} \colon \wp(\mathbb{Z}) \to \wp(\mathbb{Z}_\perp)$ and combine this with the sign-analysis image-over-approximating function obtaining a tighter $img^\sharp_{|\texttt{sum}|}$. A termination analysis describes whether the program output belongs in $\mathbb{Z}$ or $\{\perp\}$. To be more precise, we apply the analysis to a set of transformed variations of programs such that we determine whether the output may belong to $\{\perp\}$, $\mathbb{Z}$ or both for each input partition element in $T$.

The termination analysis we used for this example is AProVE [60]. Because AProVE was not intended to be used to only check a part of the input at a time, we have mechanically altered the `verb`-program slightly so that AProVE only checks for the input of interest (one element of the partition $T$ at a time). Specifically, we inserted a terminating if-expression (at the beginning of the program) that trivially excludes the other input from affecting the analysis; see Figure 5.8a. For instance, when

```
int (sum)(int x){
  if(!/* x in t */)
     {return 0;}
  int y = 0;
  while (x!= 0){
      y = y + x;
      x = x - 1;
  }
  return y;
}
```

(a)

```
int (sum)(int x){
  if(!(x=0))
     {return 0;}
  int y = 0;
  while (x!= 0){
      y = y + x;
      x = x - 1;
  }
  return y;
}
```

(b)

| $t \in T$ | $img^{\sharp}_{\lvert\texttt{sum}\rvert,\text{TERM}}$ |
|---|---|
| $\mathbb{Z}^-$ | $\{\bot\} \cup \mathbb{Z}$ |
| $\{0\}$ | $\mathbb{Z}$ |
| $\mathbb{Z}^+$ | $\mathbb{Z}$ |

(c)

Fig. 5.8: (a): An outline of an alteration of the `sum` program where `x` is in some input-partition element `t`; AProVE [60] can then decide on termination for each input-partition element $t$. (b): the alteration wherein $t = \{0\}$, *i.e.* the input `x` is zero; AProVE [60] proved that this alteration was terminating. (c): The image over-approximating function based on the termination analysis [60] (c).

checking $\texttt{x} \in \mathbb{Z}^-$ for the `sum`-program, we inserted `if(!(x<0)){ return 0;}`, as depicted in Figure 5.8b. AProVE can either prove termination, *i.e.* the output may only lie in $\mathbb{Z}$; prove nontermination, *i.e.* at least one output lies in $\{\bot\}$, such that the output may lie in $\mathbb{Z} \uplus \{\bot\}$; be unable to prove anything, *i.e.* the output may lie in $\mathbb{Z} \uplus \{\bot\}$. For the each partition element, AProVE was able to decide whether the program `sum` would yield non-terminating computations or terminating conditions, as depicted in Figure 5.8c. The resulting upper and lower probability bounds are depicted in Figure 5.9.

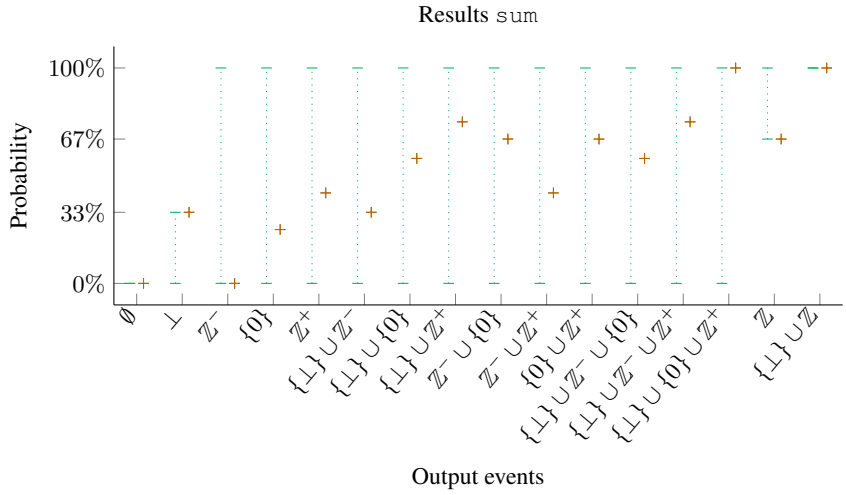



Fig. 5.9: The upper and lower probability bounds obtained by AProVE [60].

#### 5.5.3.3 Combining analyses

In this section, we give our experimental results for combining the results of the termination analysis with the (above) experimental results from the partial correctness analysis sign and achieve a new function $img^{\sharp}_{|\texttt{sum}|}$. By Lemma 5.31, we may use $img_{|\texttt{sum}|}(A) \subseteq img^{\sharp}_{|\texttt{sum}|,\mathbf{sign}}(A)$ and $img_{|\texttt{sum}|}(A) \subseteq img^{\sharp}_{|\texttt{sum}|,\text{TERM}}(A)$ to define a new over-approximating image function $img^{\sharp}_{|\texttt{sum}|}$ by

$$img^{\sharp}_{|\texttt{sum}|} \triangleq \left( img^{\sharp}_{|\texttt{sum}|,\mathbf{sign}}(A) \cap img^{\sharp}_{|\texttt{sum}|,\text{TERM}}(A) \right).$$

The result of the new image function is provided in Figure 5.10. The combination

| $t \in T$ | $img^{\sharp}_{\lvert\texttt{sum}\rvert,\text{SIGN}}$ | $img^{\sharp}_{\lvert\texttt{sum}\rvert,\text{TERM}}$ | $img^{\sharp}_{\lvert\texttt{sum}\rvert}$ |
|---|---|---|---|
| $\mathbb{Z}^-$ | $\{\bot\} \cup \mathbb{Z}^-$ | $\{\bot\} \cup \mathbb{Z}$ | $\{\bot\} \cup \mathbb{Z}^-$ |
| $\{0\}$ | $\{\bot\} \cup \{0\}$ | $\mathbb{Z}$ | $\{0\}$ |
| $\mathbb{Z}^+$ | $\{\bot\} \cup \{0\} \cup \mathbb{Z}^+$ | $\mathbb{Z}$ | $\{0\} \cup \mathbb{Z}^+$ |

Fig. 5.10: The function $img^{\sharp}_{|\texttt{sum}|}$ is the composition of two image over-approximating functions $img^{\sharp}_{|\texttt{sum}|,\text{SIGN}}$ and $img^{\sharp}_{|\texttt{sum}|,\text{TERM}}$.

improves the results, as depicted in Figure 5.11; the blue dashed lines are the bounds obtained purely by the sign analysis, and the solid black lines are the bounds obtained by the combination. We may obtain tighter bounds if we could determine that every negative input yielded an infinite computation, and, thus, obtaining $\{\bot\}$ as possible outputs for the input partition $\mathbb{Z}^-$.

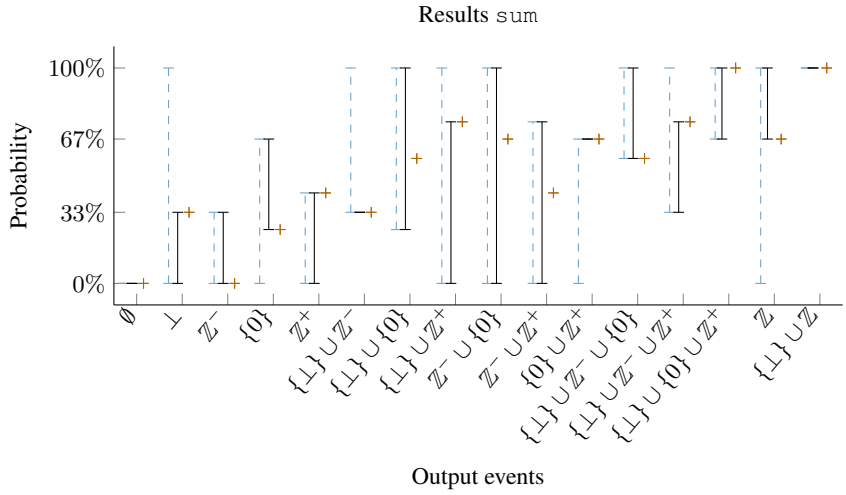



Fig. 5.11: The upper and lower bounds obtained by the sign analysis alone (blue dashed) and those bounds obtained by the combination (black solid).

## 5.6 Related work

The approaches presented in this chapter relate to the probabilistic abstract interpretation frameworks by Monniaux [95] and by Cousot and Monerau [33]. Both of these frameworks describe how to extent non-probabilistic abstract interpretation analyses for deterministic programs to probabilistic analyses for probabilistic programs. Cousot and Monerau define the extended concrete probabilistic semantics as a measurable function from the possible outputs of the random generators to the feasible concrete semantics; later, Cousot and Monerau discuss different approaches to abstracting their semantics. Monniaux takes another approach and extends, for instance, the non-probabilistic concrete semantics over environments to a new concrete semantics over probability measures (over environments). Then, he defines an abstract semantics over collections of (non-probabilistic) abstracted environments and their associated masses. Monniaux provides an example analysis, namely, the probabilistic interval analysis that we have used for comparison in the case studies. Here, we saw that the abstract transformer for the if-statement accidentally introduced an imprecision that was not caused by the original interval analysis. The approaches in this chapter transform a probability measure for a whole program and not for each program statement, unless, of course, the program consists of a single statement. When extending an analysis, it might be worth comparing the probabilistic semantics with the measure transformations for single-statement programs using the presented techniques.

In probabilistic programs, the probability of an event in general depends on (i) the inputs that may lead to the event, (ii) the probability of those inputs, and (iii) the probability that those inputs reach the event, *e.g.* random generators. Because

of (iii), *i.e.*, the random generators, both frameworks require a manual development of a new abstract semantics, *i.e.*, transformers for all types of statements; this effort allows the extended analysis to handle random generators and, thus, probabilistic programs. Our approaches handle only deterministic programs (disallowing random generators); the output event depends on (i) and (ii), and we do not need manual development; instead, we use established analysis tools to find the inputs that may lead to the event.

## 5.7 Conclusion

We have presented two techniques for reusing existing analysis to derive upper and lower probability bounds of output events. The technique for forward analysis was demonstrated by a sign analysis, an interval analysis and a termination analysis. We showed how to combine analysis results to obtain more accurate upper and lower probability bounds. The techniques are simple and pragmatic, yet the examples demonstrate their power.

## 5.8 Afterword

The probabilistic analysis approach presented in Chapter 3 could not analyse programs with non-trivial types, *e.g.*, arrays or lists. The presented techniques can analyse the same programs that the underlying analysis can analyse, *e.g.*, if the underlying analysis can analyse programs with arrays, then we can analyse those programs probabilistically as well.

We demonstrated that the forward technique handles if-expressions more precisely than Monniaux's output analysis. This is because it divides the outputs of the branches into three parts: the output that may be produced by both branches (the intersection), the output that may only be produced by the first branch, and the output that may only be produced by the second branch. This simple idea may be used in future developments of other probabilistic analyses.

The presented techniques assumed over-approximating analyses, that is, they yield functions that over-approximated the pre-images or images. By Lemma 5.3, which states the dual relation between under- and over-approximations of pre-images, the presented backward technique implicitly covers under-approximating analyses, that is, analyses that yield functions that under-approximate the pre-images. If we instead were to consider under-approximating forward analysis, we would need to introduce the concept of dual images $img^{\flat}_{f}$, that is, $img^{\flat}_{f}(A) \triangleq img^{\sharp}_{f}(A^{\complement})^{\complement}$, and investigate its relation to the pre-images $pre^{\sharp}_{f}$.

# 6

# Confluence Modulo Equivalence in Constraint Handling Rules

In this chapter, we change the focus from developing probabilistic analysis for deterministic programs to a program property for nondeterministic programs, called confluence modulo equivalence. This property is relevant since it ensures that the program's input-output relation is functional (modulo the equivalence relation) and thus that there exists a probability measure over output events. The output events must be defined as unions of sets of equivalent outputs. This property is useful for programs with redundant data representation if we are interested in upper probability bounds for a data set and not for the data representation. We study confluence modulo equivalence for nondeterministic programs written as Constraint Handling Rules programs. A program execution can be seen as a rewrite system, and we draw upon a technique from abstract reduction systems [70] that assumes program termination.

*Foreword.* The remainder of this chapter – except for the afterword (Section 6.10) – has been published with minor corrections in article [25] H. Christiansen, M. H. Kirkeby. Confluence Modulo Equivalence in Constraint Handling Rules. wolumen 8981, strony 41–58. Springer International Publishing Switzerland, 2015.

**Abstract.** Previous results on confluence for Constraint Handling Rules, CHR, are generalized to consider user-defined state equivalence relations. This allows for a much larger class of programs to enjoy the advantages of confluence, including various optimization techniques and simplified correctness proofs. A new operational semantics for CHR is introduced that significantly reduces notational overhead and allows one to consider confluence for programs with extra-logical and incomplete built-in predicates. Proofs of confluence are demonstrated for programs with redundant data representation, e.g., sets-as-lists, for dynamic programming algorithms with pruning as well as a Union-Find program, which are not covered by previous confluence concepts for CHR.

## 6.1 Introduction

A rewrite system is confluent if all derivations from a common initial state end in the same final state. Confluence, similar to termination, is often a desirable property, and

proof of confluence is a typical ingredient of a correctness proof. A programming language based on rewriting, such as Constraint Handling Rules, CHR [54, 55], ensures correctness of parallel implementations and application order optimizations.

Previous studies of confluence for CHR programs are based on Newman's lemma. This lemma concerns confluence defined in terms of alternative derivations ending in the exact same state, which excludes a large class of interesting CHR programs. However, the literature on confluence in general rewriting systems has, since the early 1970s, offered a more general concept of confluence modulo an equivalence relations. This means that alternative derivations only need to end in states that are equivalent with respect to some equivalence relation (and not necessarily identical). In this paper, we show how confluence modulo equivalence can be applied in a CHR context, and we demonstrate interesting programs covered by this concept that are not confluent by any previous definition of confluence for CHR. The use of redundant data representations is one example of what becomes within reach, and programs that search for a single best among multitudes of alternative solutions is another example.

*Example 6.1.* The following CHR program, consisting of a single rule, collects a number of separate items into a (multi-) set represented as a list of items.

```
set(L), item(A) <=> set([A|L]).
```

This rule will be applied repeatedly, replacing constraints matched by the left-hand side by those indicated to the right. The query

```
?- item(a), item(b), set([]).
```

may lead to two different final states, $\{\texttt{set([a,b])}\}$ and $\{\texttt{set([b,a])}\}$, both representing the same set. This can be formalized by a state equivalence relation $\approx$ that implies $\{\texttt{set(}L\texttt{)}\} \approx \{\texttt{set(}L'\texttt{)}\}$ whenever $L$ is a permutation of $L'$. The program is not confluent in the classical sense, as the end states are not identical, but it will be shown to be confluent modulo $\approx$.

Our generalization is based upon a new operational semantics that permits extralogical and incomplete predicates (e.g., Prolog's `var/2` and `is/2`), which is beyond the scope of previous approaches. This also leads to a noticeable reduction of notational overhead due to a simpler structure of states.

It is shown that previous results for CHR confluence, based upon critical pairs, to a large extent can be generalized for confluence modulo equivalence. We introduce additional mechanisms to handle the extra complexity caused by the equivalence relation. We do not present any (semi-) automatic approaches to confluence proofs, as this would need a formal language for specifying equivalences, which has yet to be considered.

Section 6.2 reviews previous work on confluence, in general and for CHR. Sections 6.3 and 6.4 give preliminaries and our operational semantics. Section 6.5 considers how to prove confluence modulo equivalence for CHR. Section 6.6 shows confluence modulo equivalence for a CHR version of the Viterbi algorithm; it represents a wider class of dynamic programming algorithms with pruning, also beyond

the scope of earlier proposals. Section 6.7 shows confluence modulo equivalence for the Union-Find algorithm, which has become a standard test case for confluence in CHR; it is not confluent in any previously proposed manner (except with contrived side conditions). Section 6.8 comments on related work in more detail, and the final section provides a summary and a conclusion.

## 6.2 Background

A binary *relation* $\rightarrow$ on a set $A$ is a subset of $A \times A$, where $x \rightarrow y$ denotes membership of $\rightarrow$. A *rewrite system* is a pair $\langle A, \rightarrow \rangle$; it is *terminating* if there is no infinite chain $a_0 \rightarrow a_1 \rightarrow \cdots$. The *reflexive transitive closure* of $\rightarrow$ is denoted $\overset{*}{\rightarrow}$. The *inverse relation* $\leftarrow$ is defined by $\{(y, x) \mid x \rightarrow y\}$. An *equivalence (relation)* $\approx$ is a binary relation on $A$ that is reflexive, transitive and symmetric.

A rewrite system $\langle A, \rightarrow \rangle$ is *confluent* if and only if $y \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y' \Rightarrow \exists z.\, y \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y'$, and is *locally confluent* if and only if $y \leftarrow x \rightarrow y' \Rightarrow \exists z.\, y \overset{*}{\rightarrow} z \overset{*}{\leftarrow} z'$. In 1942, Newman presented his fundamental lemma [105]: *A terminating rewrite system is confluent if and only if it is locally confluent.* An elegant proof of Newman's lemma was provided by Huet [70] in 1980.

The more general concept of *confluence modulo equivalence* was introduced in 1972 by Aho *et al*. [5] in the context of the Church-Rosser property.

**Definition 6.2 (Confluence modulo equivalence).** *A relation $\rightarrow$ is confluent modulo an equivalence $\approx$ if and only if*

$$\forall\, x, y, x', y'. \quad y \overset{*}{\leftarrow} x \approx x' \overset{*}{\rightarrow} y' \qquad \Rightarrow \qquad \exists\, z, z'. \quad y \overset{*}{\rightarrow} z \approx z' \overset{*}{\leftarrow} y'.$$

This is shown as a diagram in Fig. 6.1a. In 1974, Sethi [128] showed that confluence modulo equivalence for a bounded rewrite system is equivalent to the following properties, $\alpha$ and $\beta$, also shown in Fig. 6.1b.

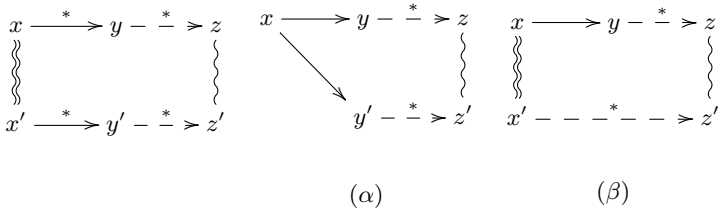
**Definition 6.3 ($\alpha$ & $\beta$).** *A relation $\rightarrow$ possesses the $\alpha$ property and the $\beta$ property if and only if it satisfies the $\alpha$ condition and the $\beta$ condition, respectively:*

$$\begin{array}{lllll} \alpha: & \forall x, y, y'. \quad y \leftarrow x \rightarrow y' & \Longrightarrow & \exists z, z'. \quad y \overset{*}{\rightarrow} z \approx z' \overset{*}{\leftarrow} y' \\ \beta: & \forall x, x', y. \quad x \approx x' \rightarrow y & \Longrightarrow & \exists z, z'. \quad x' \overset{*}{\rightarrow} z' \approx z \overset{*}{\leftarrow} y \end{array}$$

In 1980, Huet [70] generalized this result to any terminating system.

**Definition 6.4 (Local confl. mod. equivalence).** *A rewrite system is* locally confluent modulo an equivalence $\approx$ *if and only if it possesses the $\alpha$ and $\beta$ properties.*

**Theorem 6.5.** *Let $\rightarrow$ be a terminating relation. For any equivalence $\approx$, $\rightarrow$ is confluent modulo $\approx$ if and only if $\rightarrow$ is locally confluent modulo $\approx$.*




(a) Confluence modulo $\approx$. (b) Local Confluence modulo $\approx$.

Fig. 6.1: Diagrams for the fundamental concepts. A dotted arrow (single wave line) indicates an inferred step (inferred equivalence).

The known results on confluence for CHR are based on Newman's lemma. Abdennadher *et al* [3] in 1996 seem to be the first to consider this, and they showed that confluence (without equivalence) for CHR is decidable and can be checked by examining a finite set of states formed by a combination of heads of rules. A refinement, called observational confluence, was introduced in 2007 by Duck *et al* [44], in which only states that satisfy a given invariant are considered.

## 6.3 Preliminaries

We assume standard concepts of first-order logic such as predicates, atoms and terms. For any expression $E$, $vars(E)$ refers to the set of variables that occurs in $E$. A substitution is a mapping from a finite set of variables to terms, which also may be viewed as a set of first-order equations. For the substitution $\sigma$ and expression $E$, $E\sigma$ (or $E \cdot \sigma$) denotes the expression that arises when $\sigma$ is applied to $E$; the composition of two substitutions $\sigma, \tau$ is denoted $\sigma \circ \tau$. The special substitutions $failure$ and $error$ are assumed, the first representing falsity and the second representing runtime errors.

Two disjoint sets of *(user) constraints* and *built-in* predicates are assumed. For the built-in predicates, we use a semantics that is more in line with implemented CHR systems than previous approaches and that also allows for extra-logical devices such as Prolog's `var/1` and incomplete devices such as `is/2`. Whereas [3, 43, 44] collect built-in predicates in a separate store and determine their satisfiability by a magic solver that mirrors a first-order semantics, we execute a built-in predicate right away. This serves as a test, possibly producing a substitution that is immediately applied to the state.

An evaluation procedure $Exe$ for the built-in predicates $b$ is assumed such that $Exe(b)$ is either a (possibly identity) substitution to a subset of $vars(b)$ or one of $failure$ and $error$. This extends to sequences of built-in predicates as follows.

$$Exe((b_1,b_2)) = \begin{cases} Exe(b_1) & \text{when } Exe(b_1) \in \{failure, error\}, \\ Exe(b_2 \cdot Exe(b_1)) & \text{when otherwise } Exe(b_2 \cdot Exe(b_1)) \\ & \quad \in \{failure, error\}, \\ Exe(b_1) \circ Exe(b_2 \cdot Exe(b_1)) & \text{otherwise} \end{cases}$$

One subset of built-in predicates is the *logical* predicates, whose meaning is given by a first-order theory $\mathcal{B}$. For a logical atom $b$ with $Exe(b) \neq error$, the following conditions must hold.

- Partial correctness: $\mathcal{B} \models \forall_{vars(b)}(b \leftrightarrow \exists_{vars(Exe(b)) \setminus vars(b)} Exe(b))$.
- Instantiation monotonicity: $Exe(b \cdot \sigma) \neq error$ for all substitutions $\sigma$.

A logical predicate $p$ is *complete* whenever, for any atom $b$ with predicate symbol $p$, we have $Exe(b) \neq error$; later, we will define completeness with respect to a state invariant. Any built-in predicate that is not logical is called *extra-logical*. The following predicates are examples of built-in predicates; $\epsilon$ is the empty substitution.

1. $Exe(t = t') = \sigma$, where $\sigma$ is a most-general unifier of $t$ and $t'$; if no such unifier exists, the result is $failure$.
2. $Exe($`true`$)$ is $\epsilon$.
3. $Exe($`fail`$)$ is $failure$.
4. $Exe(t$ `is` $t') = Exe(t = v)$ whenever $t'$ is a ground term that can be interpreted as an arithmetic expression $e$ with the value $v$; if no such $e$ exists, the result is $error$.
5. $Exe($`var(`$t$`)`$)$ is $\epsilon$ if $t$ is a variable and $failure$ otherwise.
6. $Exe($`ground(`$t$`)`$)$ is $\epsilon$ when $t$ is ground and $failure$ otherwise.
7. $Exe(t == t')$ is $\epsilon$ when $t$ and $t'$ are identical and $failure$ otherwise.
8. $Exe(t$ `\=` $t')$ is $\epsilon$ when $t$ and $t'$ are non-unifiable and $failure$ otherwise.

The first three predicates are logical and complete; "`is`" is logical but not complete without an invariant that grounds its second arguments (considered later). The remainder are extra-logical.

The practice in previous semantics [3, 43, 44] of conjoining built-ins and testing them by satisfiability leads to ignorance of runtime errors and incompleteness.

To represent the propagation history, we introduce *indices:* An *indexed set* $S$ is a set of items of the form $x{:}i$, where $i$ belongs to some index set and each such $i$ is unique in $S$. When clear from context, we may identify an indexed set $S$ with its cleaned version $\{x \mid x{:}i \in S\}$. Similarly, the item $x$ may identify the indexed version $x{:}i$. We extend this to any structure built from indexed items.

## 6.4 Constraint Handling Rules

We define an abstract syntax of CHR together with an operational semantics suitable for considering confluence. We use the *generalized simpagation* form as a common

representation for the rules of CHR. Guards may unify variables that occur in rule bodies but not variables that occur in the matched constraints. In accordance with the standard behaviour of implemented CHR systems, failure and runtime errors are treated in the same way in the evaluation of a guard but are distinguished when occurring in a query or rule body; cf. definitions 6.6 and 6.10 below.

**Definition 6.6.** *A* rule $r$ *is of the form*

$$H_1 \setminus H_2 <=> g \mid C,$$

*where $H_1$ and $H_2$ are sequences of constraints, forming the* head *of $r$; $g$ is the* guard*, being a sequence of built-ins; and $C$ is a sequence of constraints and built-in predicates called the* body *of $r$. Either $H_1$ and $H_2$, but not both, may be empty. A* program *is a finite set of rules.*

*For any fresh variant of rule $r$ with notation as above, an* application instance $r''$ *is given as follows.*

1. *Let $r'$ be a structure of the form*
   $$H_1\tau \setminus H_2\tau <=> C\tau\sigma$$
   *where $\tau$ is a substitution for the variables of $H_1, H_2$, $Exe(g\tau) = \sigma$, $\sigma \notin \{failure, error\}$, and it holds that $(H_1 \setminus H_2)\tau = (H_1 \setminus H_2)\tau\sigma$,*
2. *$r''$ is a copy of $r'$ in which each atom in its head and body is given a unique index, where the indices used for the body are new and unused.*

*The substitution $g\tau$ is referred to as the* guard of $r''$*. The* application record *for $r''$ is a structure of the form*

$$r \mathbin{@} i_1, \ldots, i_n$$

*, where $i_1, \ldots, i_n$ is the sequence of indices of $H_1, H_2$ in the order in which they occur.*

A rule is a *simplification* when $H_1$ is empty and a *propagation* when $H_2$ is empty; in both cases, the backslash is left out, and for a propagation, the arrow symbol is written $==>$ instead. Any other rule is a *simpagation*. When the guard is the built-in $true$, it and the vertical bar may be omitted. A guard (or single built-in atom) is *logical* if it contains only logical predicates. Guards are removed from application instances as they are *a priori* satisfied. The following definition will become useful later on when we consider confluence.

**Definition 6.7.** *Consider two application instances $r_i = (A_i \setminus B_i <=> C_i)$, $i = 1, 2$. We say that $r_1$ is* blocking $r_2$ *whenever $B_1 \cap (A_2 \cup B_2) \neq \emptyset$.*

For this to be the case, $r_1$ must be a simplification or simpagation. Intuitively, this means that if $r_1$ has been applied to a state, it is not subsequently possible to apply $r_2$. In the following definition of execution states for CHR, irrelevant details of the state representation are abstracted away using the principles of [111]. To keep the notation consistent with Section 6.2, we use letters such as $x$ and $y$ to denote states.

**Definition 6.8.** *A* state representation *is a pair $\langle S, T\rangle$, where*

- *$S$ is a finite, indexed set of atoms called the* constraint store *and*
- *$T$ is a set of application records called the* propagation history.

*Two state representations $S_1$ and $S_2$ are* isomorphic*, denoted $S_1 \equiv S_2$, whenever one can be derived from the other by a renaming of variables and a consistent replacement of indices (i.e., by a 1-to-1 mapping). When $\Sigma$ is the set of all state representations, a* state *is an element of $\Sigma/_{\equiv} \cup \{failure, error\}$, i.e., an equivalence class in $\Sigma$ induced by $\equiv$ or one of two special states; applying the $failure$ ($error$) substitution to a state yields the $failure$ ($error$) state. To indicate a given state, we may for simplicity mention one of its representations.*

*A* query *$q$ is a conjunction of constraints, which is also identified with an initial state $\langle q', \emptyset \rangle$, where $q'$ is an indexed version of $q$.*

To make statements about, say, two states $x$, $y$ and an instance of a rule $r$, we may do so mentioning state representatives $x'$, $y'$ and application instance $r'$ having recurring indices. The following notions becomes useful in section 6.5, when we go into more detail on how to prove confluence modulo equivalence,

**Definition 6.9.** *An* extension *of a state $\langle S, R \rangle$ is a state of the form $\langle S\sigma \cup S^+, R \cup R^+ \rangle$ for suitable $\sigma$, $S^+$ and $R^+$; an $I$-extension is one that satisfies $I$; and a state is said to be $I$-*extendible *if it has one or more extensions that are $I$-states.*

In contrast to [3, 43, 44], we have excluded global variables, which refer to those of the original query, as they are easy to simulate: A query $q(X)$ is extended to $global('X', X), q(X)$, where $global/2$ is a new constraint predicate; $'X'$ is a constant that serves as a name of the variable. The value $val$ for $X$ is found in the final state in the unique constraint $global('X', val)$. References [3, 43, 44] use a state component for constraints waiting to be processed in addition to a separate derivation step to introduce them into the constraint store. We avoid this, as the derivations made under either premises are basically the same. Our derivation relation is defined as follows; here and in the remainder of this paper, $\uplus$ denotes the union of disjoint sets.

**Definition 6.10.** *A* derivation step $\mapsto$ *from one state to another can be of two types: by rule $\overset{r}{\mapsto}$ or by built-in $\overset{b}{\mapsto}$, defined as follows.*

*Apply:* $\langle S \uplus H_1 \uplus H_2, T \rangle \overset{r}{\mapsto} \langle S \uplus H_1 \uplus C, T' \rangle$
*whenever there is an application instance $r$ of the form $H_1 \setminus H_2 <=> C$ with $\mathrm{applied}(r) \notin T$, and $T'$ is derived from $T$ by 1) removing any application record having an index in $H_2$ and 2) adding $\mathrm{applied}(r)$ in case $r$ is a propagation.*

*Built-in:* $\langle \{b\} \uplus S, T \rangle \overset{b}{\mapsto} \langle S, T \rangle \cdot Exe(b)$.

*A state $z$ is* final *for query $q$ whenever $q \overset{*}{\mapsto} z$ and no step is possible from $z$.*

The removal of certain application records in Apply steps means to keep only those records that are essential for preventing repeated applications of the same rule to the same constraints (identified by their indices).

As noticed by [44], introducing an invariant makes more programs confluent, as one can ignore unusual states that never appear in practice. An invariant may also make it easier to characterize an equivalence relation for states.

**Definition 6.11.** *An* invariant *is a property $I(\cdot)$ that may or may not hold for a state such that for all states $x$, $y$, $I(x) \wedge (x \mapsto y) \Rightarrow I(y)$. A state $x$ for which $I(x)$ holds is called an $I$-*state*, and an $I$-*derivation *is one starting from an $I$-state. A program is $I$-*terminating *whenever all $I$-derivations are terminating. A set of* allowed queries *$Q$ may be specified, producing an invariant $reachable_Q(x) \Leftrightarrow \exists q \in Q\colon q \overset{*}{\mapsto} x$.*

*A* (state) equivalence *is an equivalence relation $\approx$ on the set of $I$-states.*

The central Theorem 6.5 applies specifically for CHR programs equipped with invariant $I$ and equivalence relation $\approx$. When $\approx$ is identity, it coincides with a theorem of [44] for observable confluence. If, furthermore, $I \Leftrightarrow true$, we obtain the classical confluence results for CHR [1].

The following definition is useful when considering confluence for programs that use Prolog built-ins such as "`is/2`".

**Definition 6.12.** *A logical predicate $p$ is* complete with respect to the invariant *$I$ (or, for short, is $I$-complete) whenever, for any atom $b$ with predicate symbol $p$ in some $I$-state, $Exe(b) \neq error$.*

A logical guard (or a built-in atom) is also called *$I$-complete* whenever all its predicates are $I$-complete. We use the term *$I$-incomplete* for any such concept that is not $I$-complete.

As promised earlier, "`is/2`" is complete with respect to an invariant that guarantees groundness of the second argument of any call to "`is/2`".

*Example 6.13.* Our semantics permits CHR programs that define constraints such as Prolog's `dif/2` constraint and a safer version of `is/2`.

```
dif(X,Y) <=> X==Y | fail.
dif(X,Y) <=> X\=Y | true.
X safer_is Y <=> ground(Y) | X is Y.
```

## 6.5 Proving Confluence Modulo Equivalence for CHR

We consider here ways to prove the local confluence properties $\alpha$ and $\beta$ from which confluence modulo equivalence may follow; cf. Theorem 6.5. The corners in the following definition generalize the critical pairs of [3]. For ease of usage, we combine the common ancestor states with the pairs, thus the concept of corners corresponding to the "given parts" of diagrams for the $\alpha$ and $\beta$ properties; cf. Fig. 6.1a. The definitions below assume a given $I$-terminating program with invariant $I$ and state equivalence $\approx$. Two states $x$ and $x'$ are *joinable modulo* $\approx$ whenever there exist states $z$ and $z'$ such that $x \overset{*}{\mapsto} z \approx z' \overset{*}{\leftarrow} x'$.

**Definition 6.14.** *An* $\alpha$-corner *consists of* $I$*-states* $x$*,* $y$ *and* $y'$ *with* $y \neq y'$ *and two derivation steps such that* $y \overset{\gamma}{\leftarrow\!\!\mapsto} x \overset{\delta}{\mapsto} y'$*. An* $\alpha$*-corner is* joinable modulo $\approx$ *whenever* $y$ *and* $y'$ *are joinable modulo* $\approx$*.*

*A* $\beta$-corner *consists of* $I$*-states* $x$*,* $x'$ *and* $y$ *with* $x \neq x'$ *and a derivation step such that* $x' \approx x \overset{\gamma}{\mapsto} y$*. A* $\beta$*-corner is joinable modulo* $\approx$ *whenever* $x'$ *and* $y$ *are joinable modulo* $\approx$*.*

The joinability of $\alpha_1$-corners holds trivially in a number of cases:

- when $\gamma$ and $\delta$ are application instances, none blocking the other;
- when $\gamma$ and $\delta$ are built-ins, both logical and $I$-complete or having no common variables; or
- when, say, $\gamma$ is an application instance whose guard is logical and $I$-complete and $\delta$ is any built-in that has no common variable with the guard of $\gamma$.

These cases are easily identified syntactically. All remaining corners are recognized as "critical", which is defined as follows.

**Definition 6.15.** *An* $\alpha$*-corner* $y \overset{\gamma}{\leftarrow\!\!\mapsto} x \overset{\delta}{\mapsto} y'$ *is* critical *whenever one of the following properties holds.*

$\alpha_1$: $\gamma$ *and* $\delta$ *are application instances where* $\gamma$ *blocks* $\delta$ *(Def. 6.7).*
$\alpha_2$: $\gamma$ *is an application instance whose guard is extra-logical or* $I$*-incomplete, and* $\delta$ *is a built-in with* $vars(g) \cap vars(\delta) \neq \emptyset$*.*
$\alpha_3$: $\gamma$ *and* $\delta$ *are built-ins with* $\gamma$ *extra-logical or* $I$*-incomplete, and* $vars(\gamma) \cap vars(\delta) \neq \emptyset$*.*

*A* $\beta$*-corner* $x' \approx x \overset{\gamma}{\mapsto} y$ *is* critical *whenever the following property holds.*

- $x \neq x'$ *and there exists no state* $y'$ *and single derivation step* $\delta$ *such that* $x' \overset{\delta}{\mapsto} y' \approx y$*.*

Our definition of critical $\beta$-corners is motivated by the experience that often the $\delta$ step can be formed trivially by applying the same rule or built-in of $\gamma$ in an analogous way to the state $x'$. By inspection and Theorem 6.5, we obtain the following.

**Lemma 6.16.** *Any non-critical corner is joinable modulo* $\approx$*.*

**Theorem 6.17.** *A terminating program is confluent modulo* $\approx$ *if and only if all its critical corners are joinable modulo* $\approx$*.*

### 6.5.1 Joinability of $\alpha_1$-critical corners

Without an invariant, an equivalence and extra-logicals, the only critical corners are of type $\alpha_1$; here, [3] has shown that the joinability of a finite set of minimal critical pairs is sufficient to ensure local confluence. In the general case, it is not sufficient to check such minimal states, but the construction is still useful as a way to group the cases that need to be considered. We adapt the definition of [3] as follows.

**Definition 6.18.** *An* $\alpha_1$-critical pattern (with evaluated guards) *is of the form*

$$\langle S_1\sigma_1, \emptyset\rangle \overset{r_1}{\leftarrowtail} \langle S, \emptyset\rangle \overset{r_2}{\mapsto} \langle S_2\sigma_2, R\rangle$$

*whenever there exist, for* $k = 1, 2$*, indexed rules* $r_k = (A_k \setminus B_k <=> g_k \mid C_k)$*, and*

$$R = \begin{cases} \{a\} & \text{whenever } r_2 \text{ is a propagation with application record } a, \\ \emptyset & \text{otherwise.} \end{cases}$$

*The remaining entities are given as follows.*

- *Let* $H_k = A_k \cup B_k$*,* $k = 1, 2$*, and split* $B_1$ *and* $H_2$ *into disjoint subsets by* $B_1 = B_1' \uplus B_1''$ *and* $H_2 = H_2' \uplus H_2''$*, where* $B_1'$ *and* $H_2'$ *must have the same number of elements* $\geq 1$*.*
- *The set of indices used in* $B_1'$ *and* $H_2'$ *are assumed to be identical, any other index in* $r_1, r_2$ *is unique, and* $\sigma$ *is a most general unifier of* $B_1'$ *and a permutation of* $H_2'$*.*
- $S = A_1\sigma \cup B_1\sigma \cup A_2\sigma \cup B_2\sigma$*, with* $S$ *being* $I$*-extendible,*
- $S_k = S \setminus B_k\sigma \cup C_k\sigma$*,* $k = 1, 2$*,*
- $g_k$ *is logical with* $\sigma_k = Exe(g_k\sigma) \notin \{error, failure\}$ *for* $k = 1, 2$*.*

*An* $\alpha_1$-critical pattern (with delayed guards) *is of the form*

$$\langle S_1, \emptyset\rangle \overset{r_1}{\leftarrowtail} \langle S, \emptyset\rangle \overset{r_2}{\mapsto} \langle S_2, R\rangle,$$

*where all parts are defined as above, except in the last step, where one of* $g_k$ *is extra-logical or its evaluation by* $Exe$ *results in* $error$*; the guards* $g_k\sigma$ *are recognized as the* unevaluated guards*.*

**Definition 6.19.** *An* $\alpha_1$*-critical corner* $y \overset{r_1}{\leftarrowtail} x \overset{r_2}{\mapsto} y'$ *is* covered by *an* $\alpha_1$*-critical pattern*

$$\langle S_1, \emptyset\rangle \overset{r_1}{\leftarrowtail} \langle S, \emptyset\rangle \overset{r_2}{\mapsto} \langle S_2, R\rangle,$$

*whenever* $x$ *is an* $I$*-extension of* $\langle S, \emptyset\rangle$*.*

Analogously to previous results on confluence of CHR [3], we can state the following.

**Lemma 6.20.** *For a given* $I$*-terminating program with invariant* $I$ *and equivalence* $\approx$*, the set of critical* $\alpha_1$*-patterns is finite, and any critical* $\alpha_1$*-corner is covered by some critical* $\alpha_1$*-pattern.*

The requirement of definition 6.18, that a critical $\alpha_1$-corner needs to be $I$-extendible, means that there may be fewer patterns to check than if classical confluence were to be investigated. Examples of this are used when showing confluence of the Union-Find program; see section 6.7 below. We can reuse the developments of [3] and the joinability results derived by their methods, e.g., using automatic checkers for classical confluence [85].

**Lemma 6.21.** *If a critical $\alpha_1$-pattern $\pi$ (viewed as an $\alpha_1$-corner) is joinable modulo the identity equivalence, then any $\alpha_1$-corner covered by $\pi$ is joinable under any $I$ and $\approx$.*

This means that we may succeed in showing confluence modulo $\approx$ under $I$ in the following way for a program without critical $\alpha_2$, $\alpha_3$ and $\beta$ corners.

- Run a classical confluence checker (e.g., [85]) to identify which classical, critical pairs are not joinable. Those that do not correspond to $I$-extendible $\alpha_1$ patterns can be disregarded.
- The critical $\alpha_1$-patterns that remain need separate proofs, which may succeed due to the stronger antecedent given by $I$ and the weakening of the joinability consequent by an equivalence relation.

*Example 6.22 (example 6.1, continued).* We consider again the one-line program of example 6.1 that collects items into a set, represented as a list. Suitable invariant and equivalence are given as follows; the propagation history can be ignored, as there are no propagations.

$I$: $I(x)$ holds if and only if $x = \{\texttt{set}(L)\} \cup \mathit{Items}$, where $\mathit{Items}$ is a set of `item/1` constraints whose argument is a constant and $L$ a list of constants.
$\approx$: $x \approx x'$ if and only if $x = \{\texttt{set}(L)\} \cup \mathit{Items}$ and $x' = \{\texttt{set}(L')\} \cup \mathit{Items}$, where $\mathit{Items}$ is a set of `item/1` constraints and $L$ is a permutation of $L'$.

There are no built-ins and thus no critical $\alpha_2$- or $\alpha_3$-patterns. There is only one critical $\alpha_1$-pattern, namely,

$$\{\texttt{set([B|L])}, \texttt{item(A)}\} \leftarrow\!\!\!\shortmid \{\texttt{set(L)}, \texttt{item(A)}, \texttt{item(B)}\} \mapsto \{\texttt{set([A|L])}, \texttt{item(B)}\}.$$

The participating states are not $I$-states, as `A`, `B` and `L` are variables; the set of all critical $\alpha_1$-corners can be generated by different instantiations of the variables, discarding those that lead to non-$I$-states. We cannot use Lemma 6.21 to prove joinability, as the equivalence is $\approx$ essential. Instead, we can apply a general argument that holds for any $I$-extension of this pattern. The common ancestor state in such an $I$-extension is of the form $\{\texttt{set}(L), \texttt{item}(A)\} \cup \mathit{Items}$, and joinability is shown by applying the rule to the two "wing" states (not shown) to form the two states $\{\texttt{set}([\text{B},\text{A},|\text{L}])\} \cup \mathit{Items} \approx \{\texttt{set}([\text{A},\text{B},|\text{L}])\} \cup \mathit{Items}$. To show confluence modulo $\approx$, we still need to consider the $\beta$-corners, which we return to in example 6.24 below.

### 6.5.2 About critical $\alpha_2$-, $\alpha_3$- and $\beta$-corners

It is not possible to characterize the sets of all critical $\alpha_2$-, $\alpha_3$- and $\beta$-corners by finite sets of patterns of mini-states in the same way as for $\alpha_1$.

The problem for $\alpha_2$ and $\alpha_3$ stems from the presence of extra-logical or incomplete built-ins. Here, the existence of one derivation step from a given state $S$ does not imply the existence of another, analogous derivation step from an extension $S\sigma \cup S^+$. This is demonstrated by the following example.

*Example 6.23.* Consider the following program that has extra-logical guards.

```
r1: p(X) <=> var(X)    | q(X).
r2: p(X) <=> nonvar(X) | r(X).
r3: q(X) <=> r(X).
```

There are no propagation rules; thus, we can identify states with multisets of constraints. The invariant $I$ is given as follows, and the state equivalence is trivial identity; thus, there are no critical $\beta$-corners to consider.

$I(S)$: $S$ is a multiset of `p`, `q` and `r` constraints and built-ins formed by the "=" predicate. Any argument is either a constant or a variable.

The meaning of equality built-ins is as defined in section 6.3 above.

It can be argued informally that this program is $I$-confluent, as all user-defined constraints will eventually become `r` constraints unless a failure occurs due to the execution of equality built-ins. The latter can only be introduced in the initial query, and thus, if one derivation leads to failure, all terminated derivations do. Termination follows from the inherent stratification of the constraints.

To prove this formally, we consider all critical corners and demonstrate that they are joinable. One group of critical $\alpha_2$-corners is of the following form: (1)

$$S_1 = \ \big(\{\mathtt{q}(x), x = a\} \uplus S\big) \overset{r_1}{\leftarrowtail} \big(\{\mathtt{p}(x), x = a\} \uplus S\big) \overset{=}{\mapsto} \big(\{\mathtt{p}(a)\} \uplus S\big) \ = S_2;$$

$x$ is a variable, $a$ is a constant, and $S$ is an arbitrary set of constraints such that $I$ is maintained. Any such corner is joinable, which can be shown as follows: (2)

$$S_1 \overset{=}{\mapsto} S_1' \overset{r_2}{\mapsto} \{\mathtt{r}(a)\} \uplus S \overset{r_2}{\leftarrowtail} S_2;$$

The remaining critical $\alpha_2$-corners form a similar group.

$$\{\mathtt{q}(x), x = y\} \uplus S \overset{r_1}{\leftarrowtail} \{\mathtt{p}(x), x = y\} \uplus S \overset{=}{\mapsto} \{\mathtt{p}(x)\} \uplus S;$$

$x$ and $y$ variables, $r_1$ and $S$ and $S$ an arbitrary set of constraints such that $I$ is maintained. Joinability is shown by a similar argument that goes for this entire group. The only critical corners are those $\alpha_2$ cases that have been considered, and thus, the program is confluent.

We notice, however, that the derivation steps in (1) and (2) are possible only due to the assumptions about the permitted instances of $x$, $a$ and $S$. The symbol $a$, for example, is not a variable in a formal sense, neither is it a constant; rather, it is a meta-variable or placeholder of the sort that mathematicians use all the time. This means that we cannot reduce formulas (1) and (2) to refer to derivations over mini-states, with proper variables as placeholders, as then $r_2$ can never apply.

To see critical $\alpha_3$-corners, we change $I$ into $I'$ by also allowing `var` constraints in a state. One group of such corners will have the following shape.

$$\{\mathtt{var}(a)\} \uplus S \overset{=}{\leftarrowtail} \{\mathtt{var}(x), x = a\} \uplus S \overset{\mathtt{var}}{\mapsto} \{x = a\} \uplus S$$

$x$ is a variable, $a$ is a constant, and $S$ is an arbitrary set of constraints such that $I'$ is maintained. For, e.g., $S = \emptyset$, this corner is obviously not joinable, and thus, the program is not confluent (module equivalence) under $I'$. As above, we observe that the set of critical $\alpha_3$ corners cannot be characterized by a finite set of mini-states.

The $\beta$ property needs to be considered when the state equivalence is non-trivial, as in the following example

*Example 6.24 (examples 6.1 and 6.22, continued).* To check the $\beta$ property, we notice that any $\beta$-corner is of the form

$$\{\texttt{set(}L'\texttt{),item(}A\texttt{)}\} \uplus \mathit{Items} \approx \{\texttt{set(}L\texttt{),item(}A\texttt{)}\} \uplus \mathit{Items} \mapsto \{\texttt{set([}A\texttt{|}L\texttt{])}\} \uplus \mathit{Items}$$

where $L$ and $L'$ are lists, one being a permutation of the other. Applying the rule to the "left wing" state leads to $\{\texttt{set([}A\texttt{|}L'\texttt{])}\} \cup \mathit{Items}$, which is equivalent (wrt. $\approx$) to the "right wing" state; there are thus no critical $\beta$-corners. Together with the results for the critical $\alpha$-corners above, we have now shown local confluence modulo $\approx$ for the sets-as-lists program, and as the program is clearly $I$-terminating, it follows that it is confluent modulo $\approx$.

## 6.6 Confluence of Viterbi Modulo Equivalence

Dynamic programming algorithms produce solutions to a problem by generating solutions to a subproblem and iteratively extending the subproblem and its solutions (until the original problem is solved). The Viterbi algorithm [138] finds a most probable path of state transitions in a Hidden Markov Model (HMM) that produces a given emission sequence $Ls$, also called the *decoding* of $Ls$; see [45] for some background on HMMs. There may be exponentially many paths, but an early pruning strategy ensures linear time. The algorithm has been studied in CHR by [24], starting from the following program; the "@" operator is part of the implemented CHR syntax used for labelling rules.

```
:- chr_constraint path/4, trans/3, emit/3.

expand @ trans(Q,Q1,PT),
         emit(Q,L,PE),
         path([L|Ls],Q,P,PathRev) ==>
         P1 is P*PT*PE  |  path(Ls,Q1,P1,[Q1|PathRev]).

prune @ path(Ls,Q,P1,_) \
        path(Ls,Q,P2,_)   <=>
        P1 >= P2 | true.
```

The meaning of a constraint `path(`$Ls$`,`$q$`,`$p$`,`$R$`)` is that $Ls$ is a remaining emission sequence to be processed, $q$ is the current state of the HMM, and $p$ is the probability of a path $R$ found for the already processed prefix of the emission sequence. To simplify the program, a path is represented in reverse order. Constraint

`trans(`$q$`,`$q'$`,`$pt$`)` indicates a transition from state $q$ to state $q'$ with probability $pt$, and `emit(`$q$`,`$\ell$`,`$pe$`)` indicates a probability $pe$ for emitting letter $\ell$ in state $q$.

The decoding of a sequence $Ls$ is stated by a query of the form "***HMM,*** `path(`$Ls$`,q0,1,[])`", where ***HMM*** is an encoding of a particular HMM in terms of `trans` and `emit` constraints. Assuming ***HMM*** and $Ls$ to be fixed, the state invariant $I$ is given as reachability from the indicated query. The program is $I$-terminating, as any new `path` constraint introduced by the expand rule has a first argument shorter than that of its predecessor. Depending on the application order, it may run in between linear and exponential time, and [24] proceeds by semantics-preserving program transformations that lead to an optimal execution order.

The program is not confluent in the classical sense, i.e., without an equivalence, as the `prune` rule may need to select one of two different and equally probable paths. A suitable state equivalence may be defined as follows.

**Definition 6.25.** *Let* $\langle$***HMM*** $\cup$ ***PATHS***$_1, T\rangle \approx \langle$***HMM*** $\cup$ ***PATHS***$_2, T\rangle$ *whenever: For any indexed constraint* $(i\colon$ `path(`$Ls$`,`$q$`,`$P$`,`$R_1$`)` $) \in$ ***PATHS***$_1$*, there is a corresponding* $(i\colon$ `path(`$Ls$`,`$q$`,`$P$`,`$R_2$`)` $) \in$ ***PATHS***$_2$ *and vice versa.*

The built-ins used in guards, `is/2` and `>=/2`, are logical and $I$-complete, and thus, there are no $\alpha_2$- or $\alpha_3$-critical corners. For simplicity of notation, we ignore the propagation histories. There are three critical $\alpha_1$ patterns to consider:
*(i)* $y \overset{\texttt{prune}}{\longleftarrow} x \overset{\texttt{prune}}{\longmapsto} y'$, where $x$ contains two `path` constraints that may differ only in their last arguments, and $y$ and $y'$ differ only in which of these constraints are preserved; thus, $y \approx y'$.
*(ii)* $y \overset{\texttt{prune}}{\longleftarrow} x \overset{\texttt{expand}}{\longmapsto} y'$ where $x = \{\pi_1, \pi_2, \tau, \eta\}$, $\pi_i =$ `path(`$L$`,`$q$`,`$P_i$`,`$R_i$`)` for $i = 1, 2$, $P_1 \geq P_2$, and $\tau, \eta$ the `trans` and `emit` constraints used for the expansion step. Thus, $y = \{\pi_1, \tau, \eta\}$ and $y' = \{\pi_1, \pi_2, \pi_2', \tau, \eta\}$, where $\pi_2'$ is expanded from $\pi_2$. To show joinability, we show the stronger property of the existence of a state $z$ with $y \overset{*}{\longmapsto} z \overset{*}{\longleftarrow} y'$. We select $z = \{\pi_1, \pi_1', \tau, \eta\}$, where $\pi_1'$ is expanded from $\pi_1$.[1] The probability in $\pi_1'$ is greater than or equal to that of $\pi_2'$, which means that a pruning of $\pi_2'$ is possible when both are present. Joinability is shown as follows.

$$y \overset{\texttt{expand}}{\longmapsto} z \overset{\texttt{prune}}{\longleftarrow} \{\pi_1, \pi_1', \pi_2, \tau, \eta\} \overset{\texttt{prune}}{\longleftarrow} \{\pi_1, \pi_1', \pi_2, \pi_2', \tau, \eta\} \overset{\texttt{expand}}{\longleftarrow} y'$$

*(iii)* As case *ii* but with $P_2 \geq P_1$ and $y = \{\pi_2, \tau, \eta\}$; the proof is similar and therefore omitted.

Thus, all $\alpha$-critical corners are joinable. There are no critical $\beta$ corners, as whenever $x' \approx x \overset{r}{\longmapsto} y$, the rule $r$ can apply to $x'$ with an analogous result, i.e., there exists a state $y'$ such that $x' \overset{r}{\longmapsto} y' \approx y$. This finishes the proof of confluence modulo $\approx$.

[1] It may be the case that $\pi_1'$ was produced and pruned at an earlier stage, and thus, the propagation history prevents the creation of $\pi_1'$ anew. A detailed argument can show that in this case, there will be another constraint $\pi_1''$ in store similar to $\pi_1'$ but with a $\geq$ probability, and $\pi_1''$ can be used for pruning $\pi_2'$ and obtaining the desired result in that manner.

## 6.7 Confluence of Union-Find Modulo Equivalence

The Union-Find algorithm [133] maintains a collection of disjoint sets under union, with each set represented as a tree. It has been implemented in CHR by [126], who proved that it is nonconfluent using critical pairs [3]. We have adapted a version from [44], extending it with a new `token` constraint to be explained. Let $UF_{\texttt{token}}$ refer to our program, and let $UF_0$ refer to the original without `token` constraints.

```
union    @ token, union(A,B) <=>
                        find(A,X), find(B,Y), link(X,Y).
findNode @ A ~> B \ find(A,X) <=> find(B,X).
findRoot @ root(A) \ find(A,X) <=> A=X.
linkEq   @ link(A,A) <=> token.
link     @ root(A) \ link(A,B), root(B) <=> B ~> A, token.
```

The `~>` and `root` constraints, called *tree constraints*, represent a set of trees. A finite set $T$ of ground tree constraints is *consistent* whenever the following is true: for any constant $a$ in $T$, there is either one and only one `root(`$a$`)` $\in T$ or $a$ is connected via a unique chain of `~>` constraints to some $r$ with `root(`$r$`)` $\in T$. We define $sets(T)$ to be the set of sets represented by $T$, formally: the smallest equivalence relation over constants in $T$ that contains the reflexive and transitive closure of `~>`; $set(a, T)$ refers to the set in $sets(T)$ containing the constant $a$.

The *allowed queries* are ground and of the form $T \cup U \cup \{\texttt{token}\}$, where $T$ is a consistent set of tree constraints and $U$ is a set of constraints `union(`$a_i$`,`$b_i$`)`, where $a_i, b_i$ appear in $T$. The `token` constraint is necessary for triggering the `union` rule, and thus, it needs to be present in the query to get the process started. It is consumed when one `union` operation starts and is reintroduced when it has finished (as marked by the `linkEq` or `link` rules), thus ensuring that no two `union` operations overlap in time. The invariant $I$ is defined by reachability from these queries. By induction, we can show the following properties of any $I$-state $S$.

- Either $S = T \cup U \cup \{\texttt{token}\}$, where $T$ is a consistent set of tree constraints and $U$ is a set of `union` constraints whose arguments are in $T$, or
- $S = T \cup U \cup \{\texttt{link(}A_1\texttt{,}A_2\texttt{)}\} \cup F_1 \cup F_2$, where $T, U$ are as in the previous case, and for $i = 1, 2$,
  - if $A_i$ is a constant, $F_i = \emptyset$; otherwise,
  - $F_i = \{\texttt{find(}a_i\texttt{,}A_i\texttt{)}\}$ or $F_i = \{(a_i = A_i)\}$ for some constant $a_i$.

As shown by [126], $UF_0$ is not confluent in the classical sense and can be related to the following issues.

*(i)* When the detailed steps of two `union` operations are intertwined in an unfortunate way, the program may become stuck in a state wherein it cannot finish the operation, as shown in the following derivation.
{`root(a), root(b), root(c), union(a,b), union(b,c)`} $\overset{*}{\mapsto}$
{`root(a), root(b), root(c), link(a,b), link(b,c)`} $\mapsto$
{`b ~> a, root(a), root(c), link(b,c)`}

*(ii)* Different execution orders of the `union` operations may lead to different data structures (representing the same sets) as exemplified by the following derivations from a query $q_0$.

$q_0 = \{\texttt{root(a)}, \texttt{root(b)}, \texttt{root(c)}, \texttt{union(a,b)}, \texttt{union(b,c)}\}$.
$q_0 \overset{*}{\mapsto} \{\texttt{root(a)}, \texttt{root(c)}, \texttt{b ~> a}, \texttt{union(b,c)}\}$
$\quad \overset{*}{\mapsto} \{\texttt{root(a)}, \texttt{b ~> a}, \texttt{c ~> a}\}$
$q_0 \overset{*}{\mapsto} \{\texttt{root(a)}, \texttt{root(b)}, \texttt{c ~> b}, \texttt{union(a,b)}\}$
$\quad \overset{*}{\mapsto} \{\texttt{root(b)}, \texttt{b ~> a}, \texttt{c ~> b}\}$

We proceed to show that $UF_{\texttt{token}}$ is confluent modulo an equivalence $\approx$, defined as follows; the letters $U$ and $T$ refer to sets of `union` constraints and sets of tree constraints.

- $T \cup U \cup \{\texttt{token}\} \approx T' \cup U \cup \{\texttt{token}\}$ whenever $sets(T) = sets(T')$.
- $T \cup U \cup \{\texttt{link}(A_1, A_2)\} \cup F_1 \cup F_2 \approx T' \cup U \cup \{\texttt{link}(A_1', A_2')\} \cup F_1' \cup F_2'$ whenever $sets(T) = sets(T')$, and for $i = 1, 2$, that
  - if $A_i$ is a constant and (by $I$) $F_i = \emptyset$, then $A_i'$ is a constant, $set(A_i, T) = set(A_i', T')$ and $F_i' = \emptyset$
  - if $A_i$ is a variable and $F_i = \{\texttt{find}(a_i, A_i)\}$ for some constant $a_i$, then $F_i' = \{\texttt{find}(a_i', A_i')\}$ and $set(a_i, T) = set(a_i', T')$,
  - if $A_i$ is a variable, $F_i = \{(a_i = A_i)\}$ for some constant $a_i$ with $\texttt{root}(a_i) \in T$ then $F_i' = (a_i' = A_i')\}$, $\texttt{root}(a_i') \in T'$ and $set(a_i, T) = set(a_i', T')$.

There are no critical $\alpha_2$- or $\alpha_3$-patterns. The $\alpha_1$-patterns (critical pairs) of $UF_{\texttt{token}}$ are those of $UF_0$ and a new one, formed by an overlap of the `union` rule with itself, as shown below. We reuse the analysis of [126], who identified all critical pairs for $UF_0$; by Lemma 6.21, we consider only those pairs, and they are identified as non-joinable.

In [126], eight non-joinable critical pairs are identified; the first pair ("the unavoidable" pair) concerns issue *(ii)*. Its ancestor state $\{\texttt{find}(B, A), \texttt{root}(B), \texttt{root}(C), \texttt{link}(C, B)\}$, is excluded by $I$: when any corner is covered, $B$ and $C$ must be ground in addition to the `link` constraint, which according to $I$ excludes a `find` constraint. This can be traced to the effect of our `token` constraint, which forces any `union` to complete its detailed steps before the next `union` may be entered. However, issue *(ii)* arises in the new $\alpha_1$-pattern for $UF_{\texttt{token}}$, $y \leftarrow\!\!\shortmid\ x \mapsto y'$ where

$$
\begin{aligned}
x &= \{\texttt{token}, \texttt{union}(A, B), \texttt{union}(A', B')\} \\
y &= \{\texttt{find}(A, X), \texttt{find}(B, Y), \texttt{link}(X, Y), \texttt{union}(A', B')\} \\
y' &= \{\texttt{find}(A', X'), \texttt{find}(B', Y'), \texttt{link}(X', Y'), \texttt{union}(A, B)\}
\end{aligned}
$$

Showing the joinability of any corner covered by this pattern means to find $z, z'$ such that $y \overset{*}{\mapsto} z \approx z' \overset{*}{\leftarrow\!\!\shortmid} y'$. This can be done by, from $y$, first executing all remaining steps related to $\texttt{union}(A, B)$ and then the steps relating to $\texttt{union}(A', B')$ to reach a state $z = T \cup U \cup \{\texttt{token}\}$. In a similar manner, we construct $z' = T' \cup U \cup \{\texttt{token}\}$, starting with the steps relating to $\texttt{union}(A', B')$ followed by those of $\texttt{union}(A, B)$. It can be proved by induction that $sets(T) = sets(T')$; thus, $z \approx z'$.

Next, [126] identifies three critical pairs, which imply inconsistent tree constraints. The authors argue informally that these pairs will never occur for a query with consistent tree constraints. As noticed by [44], this can be formalized using an invariant. The last four pairs of [126] relate to issue *(i)* above; [126] argues these to be avoidable, referring to procedural properties of implemented CHR systems (which is slightly unusual in a context concerning confluence). In [44], those pairs are avoided by restricting allowed queries to include only a single `union` constraint; we can allow any number of those, but we avoid the problem due to the control patterns imposed by the `token` constraints and formalized in our invariant $I$.

This completes the argument that $UF_{\texttt{token}}$ satisfies the $\alpha$ property, and by inspection of the possible derivation steps one by one (for each rule and for the "=" constraint), it can be seen that there are no critical $\beta$ corners. Thus, $UF_{\texttt{token}}$ is locally confluent modulo $\approx$, and since tree consistency implies termination, it follows that $UF_{\texttt{token}}$ is confluent modulo $\approx$.

## 6.8 Discussion and detailed comments on related work

We already commented on the foundational work on confluence for CHR by [3], who, with reference to Newman's lemma, devised a method to prove confluence by inspecting a finite number of critical pairs. This also formed the foundation of automatic confluence checkers [3, 43, 85] (with no invariant or equivalence).

The addition of an invariant $I$ in the specification of confluence problems for CHR was suggested by [44]. The authors considered a construction similar to our $\alpha_1$-corners and critical $\alpha_1$-patterns. They noted that critical $\alpha_1$-patterns usually do not satisfy the invariant, and thus, they based their approach on defining a collection of corners based on $I$-states as minimal extensions of such patterns. Local confluence, then, follows from the joinability of this collection of minimally extended states. However, there are often infinitely many such minimally extended states; this occurs even for a natural invariant, such as groundness, when infinitely many terms are possible, as is the case in Prolog-based CHR versions. We can use this construction (in cases where it is finite!) to further cluster the space of our critical corners, but our examples worked quite well without this.

Of other work concerned with confluence for CHR, we can mention [63, 112], which considered confluence for non-terminating CHR programs. We can also refer to [131], which gives an overview of CHR-related research until 2010, including confluence.

## 6.9 Conclusion and future work

We have introduced confluence modulo equivalence for CHR, which allows for a much larger class of programs to be characterized as confluent in a natural way, thus increasing the practical relevance of confluence for CHR.

We demonstrated the power of the framework by showing confluence modulo equivalence for programs that use a redundant data representation (the set-as-lists and Union-Find programs) and a dynamic programming algorithm (the Viterbi program); all these are beyond the scope of previous confluence concepts for CHR. With the new operational semantics, we can also handle extra-logical and incomplete built-in predicates, and the notational improvements obtained by this semantics may also promote new applications of and research on confluence.

As a first steps towards semi- or fully automatic proof methods, it is important to notice that the classical joinability of a critical pair – as can be decided by existing confluence checkers, such as [85] – provide a sufficient condition for joinability modulo any equivalence. Thus, only classically non-joinable pairs – in our terminology, $\alpha_1$ patterns – need to be examined in more details involving the relevant equivalence; however, in some cases, there may also be critical $\alpha_2$, $\alpha_3$ and $\beta$ patterns that need to be considered.

While the set of critical $\alpha_1$-patterns can be characterized by a finite collection of patterns consisting of mini-states tied together by derivations, the same things are not possible for other types of critical patterns. In our examples, we used semi-formal patterns, whose meta-variables or placeholders are covered by side conditions such as "$x$ is a variable" and "$a$ is a constant". However, this must be formalized to approach automatic or semi-automatic methods. A formal and machine-readable language for specifying invariants and equivalences will also be an advantage in this respect.

## 6.10 Afterword

In this chapter, we introduced confluence modulo equivalence in the context of constraint-handling rules and demonstrated the power of the framework by showing confluence modular equivalents for two data redundant programs and one dynamic programming algorithm. We have continued this work in [26], where we consider the semantic requirements for constructing the proofs mechanically.

Returning to the context of probabilistic analysis, confluence modulo equivalence implies a functional relation from input to non-overlapping sets of equivalent output. In case the output properties of interest can be constructed by a union of these sets/events, there exists a probability measure over these output properties. In this case we may use the presented approaches, *e.g.* those of Chapter 5.

# 7

# Confluence and Convergence in Probabilistically Terminating Reduction Systems

In this chapter, we focus on a property for probabilistic programs that reduces their semantics to functions, which in return ensures that there exists an output probability measure. The property is called almost-sure convergence, and it ensures that, for each input, the program terminates in the same state with probability one. We consider probabilistic programs in the form of probabilistic abstract reduction systems [19] and show that such a program is almost sure convergent if and only if it is confluent and almost-surely terminating, that is, the program terminates with probability one.

*Foreword.* The remainder of this chapter – except for the afterword (Section 7.8) – has been published with minor corrections in article [77] M. H. Kirkeby, H. Christiansen. Confluence and convergence in probabilistically terminating reduction systems. *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017*, wolumen abs/1709.05123, 2017. (accepted for publication).

**Abstract.** Convergence of an abstract reduction system (ARS) is the property that any derivation from an initial state will end in the same final state, a.k.a. normal form. We generalize this for probabilistic ARS as almost-sure convergence, meaning that the normal form is reached with probability one, even if diverging derivations may exist. We show and exemplify properties that can be used for proving the almost-sure convergence of probabilistic ARS, generalizing known results from ARS.

## 7.1 Introduction

Probabilistic abstract reduction systems, PARS, are general models of systems that develop over time in discrete steps [20]. In each non-final state, the choice of successor state is governed by a probability distribution, which in turn induces a global, probabilistic behaviour of the system. Probabilities make termination more than a simple yes-no question, and the following criteria have been proposed: *probabilistic termination* – a derivation terminates with some probability $> 0$ – and *almost-sure*

*termination* – a derivation terminates with probability $= 1$, even if infinite derivations may exist (and whose total probability thus amounts to 0). When considering a PARS as a computational system, almost-sure termination may be the most interesting, and there exist well-established methods for proving this property [19, 50].

PARS covers a variety of probabilistic algorithms and programs, scheduling strategies and protocols [13, 20, 110], and PARS is a well-suited abstraction level for better understanding their termination and correctness properties. Randomized and probabilistic algorithms (e.g., [12, 89, 102]) can be classified into two groups: Monte Carlo Algorithms, which allow a set of alternative outputs (typically only correct with a certain probability or within a certain accuracy), e.g., Karger-Stein's Minimum Cut [75], Monte Carlo integration and Simulated Annealing [79], and Las Vegas Algorithms, which provide one (correct) output and which may be both simpler and on average more efficient than their deterministic counterparts, e.g., Randomized Quicksort [53], checking equivalence of circular lists [71], and probabilistic modular GCD [148]. We focus on results that are relevant for the latter type of systems, and here, the property of *convergence* is interesting, as it may be a necessary condition for correctness: a system is convergent if it is guaranteed to terminate with a unique result. We introduce the concept of *almost-sure convergence* for PARS, meaning that a unique result is found with probability $= 1$, although there may be diverging computations; this property is a necessary condition for partial correctness, more precisely, a strengthened version of partial correctness whereby the probability of not obtaining a result is zero. The related concept of *confluence* has been extensively studied for ARS, e.g., [11, 70], and especially for terminating systems for which confluence implies convergence: a system is confluent if, whenever alternative paths (i.e., repeated reductions and computations) are possible from some state, these paths can be extended to join in a common state. Newman's lemma [105] from 1942 is one of the most central results: in a terminating system, confluence (and thus convergence) can be shown from a simpler property called local confluence. In, e.g., term rewriting [11] and (a subset of) the programming language CHR [1, 3], proving local confluence may be reduced to a finite number of cases, described by *critical pairs* (for a definition, see these references), which in some cases may be checked automatically. It is well-known that Newman's lemma does not generalize to non-terminating systems (and thus neither to almost-sure terminating systems); see, e.g., [70].

Probabilistic and almost-sure versions of confluence were introduced concurrently by Frühwirth et al. [56] – in the context of a probabilistic version of CHR – and by Bournez and Kirchner [20] in more generality for PARS. However, the definitions in the latter reference were given indirectly, assuming an insight into Homogeneous Markov Chain Theory, and a number of central properties were listed without hints of proofs.

In the present paper, we consider the important property of almost-sure convergence for PARS and state properties that are relevant for proving it. In contrast to [20], our definitions are self-contained, being based on elementary math, and proofs are included. One of our main and novel results is that almost-sure termination together with confluence (in the classical sense) gives almost-sure conver-

gence. Almost-sure convergence and almost-sure termination were introduced in an early 1983 paper [66] for a specific class of probabilistic programs with a finite state space, but our generalization to PARS appears to be new.

In 1991, Curien and Ghelli [34] described a powerful method for proving confluence of non-probabilistic systems, therein using suitable transformations from the original system into one, known to be confluent. We can show how this result applies to probabilistic systems, and we develop an analogous method for also proving non-confluence.

In section 7.2, we review definitions for abstract reduction systems and introduce and motivate our choices of definitions for their probabilistic counterparts; a proof that the defined probabilities actually constitute a probability distribution is found in the Appendix. Section 7.3 formulates and proves important properties, relevant for showing almost-sure convergence of particular systems. Section 7.4 goes in detail with applications of the transformational approach [34] to (dis-) proving almost-sure convergence, and in Section 7.5, we demonstrate the use of this for a random walk system and Hermans' Ring. We add a few more comments on selected, related work in section 7.6, and section 7.7 provides a summary and suggestions for future work.

## 7.2 Basic definitions

The definitions for non-probabilistic systems are standard; see, e.g., [11, 70].

**Definition 7.1 (ARS).** *An* Abstract Reduction System *is a pair* $R = (A, \to)$ *whereby the* reduction $\to$ *is a binary relation on a countable set* $A$.

Instead of $(s,t) \in \to$, we write $s \to t$ (or $t \leftarrow s$ when convenient), and $s \to^* t$ denotes the transitive reflexive closure of $\to$.

In the literature, an ARS is often required to have only finite branching, i.e., for any element $s$, the set $\{t \mid s \to t\}$ is finite. We do not require this, as the implicit restriction to countable branching is sufficient for our purposes.

The set of *normal form*s $R_{NF}$ are those $s \in A$ for which there is no $t \in A$ such that $s \to t$. For a given element $s$, the *normal forms of* $s$ are defined as the set $R_{NF}(s) = \{t \in R_{NF} \mid s \to^* t\}$. An element that is not a normal form is said to be *reducible*, i.e., an element $s$ is reducible if and only if $\{s' \mid s \to s'\} \neq \emptyset$.

A *path* from an element $s$ is a (finite or infinite) sequence of reductions $s \to s_1 \to s_2 \to \cdots$. A finite path $s \to s_1 \to s_2 \to \cdots \to s_n$ has *length* $n$ ($n \geq 0$); in particular, we recognize an empty path (of length 0) from a given state to itself. For given elements $s$ and $t \in R_{NF}(s)$, $\Delta(s,t)$ denotes the set of finite paths $s \to \cdots \to t$ (including the empty path); $\Delta^\infty(s)$ denotes the set of infinite paths from $s$. A system is

- *confluent* if for all $s_1 \leftarrow^* s \to^* s_2$ there is a $t$ such that $s_1 \to^* t \leftarrow^* s_2$,
- *locally confluent* if for all $s_1 \leftarrow s \to s_2$ there is a $t$ such that $s_1 \to^* t \leftarrow^* s_2$,
- *terminating*[1] iff it has no infinite path,

[1] A terminating system is also called *strongly normalizing* elsewhere, e.g., [34].

- *convergent* iff it is terminating and confluent, and
- *normalizing*[2] iff every element $s$ has a normal form, i.e., there is an element $t \in R_{NF}$ such that $s \to^* t$.

Notice that a normalizing system may not be terminating. A fundamental result for ARS is Newman's Lemma: a terminating system is confluent if and only if it is locally confluent.

The following property indicates the complexity of the probability measures that are needed to address paths in probabilistic abstract reduction systems defined over countable sets.

**Proposition 7.2.** *Given an ARS as above and given elements $s$ and $t \in R_{NF}(s)$, it holds that $\Delta(s,t)$ is countable, and $\Delta^\infty(s)$ may or may not be countable.*

*Proof.* For the first part, $\Delta(s,t)$ is isomorphic to a subset of $\bigcup_{n=1,2,\ldots} A^n$. A countable union of countable sets is countable, and thus, $\Delta(s,t)$ is countable.

For the second part, consider the ARS $\langle \{0,1\}, \{i \to j \mid i,j \in \{0,1\}\}\rangle$. Each infinite path can be read as a real number in the unit interval, and any such real number can be described by an infinite path. The real numbers are not countable.

This means that we can define discrete and summable probabilities over $\Delta(s,t)$, and – which we will avoid – considering probabilities over the space $\Delta^\infty(s)$ requires a more advanced measure.

In the next definition, a path is considered a Markov process/chain, i.e., each reduction step is independent of the previous steps, and thus, the probability of a path is defined as a product in the usual way. PARS can be seen as a special case of Homogeneous Markov Chains, cf. [20], but for practical reasons, it is relevant to introduce them as generalizations of ARS.

**Definition 7.3 (PARS).** *A* Probabilistic Abstract Reduction System *is a pair $R^P = (R,P)$ whereby $R = (A,\to)$ is an ARS, and for each reducible element $s \in A \backslash R_{NF}$, $P(s \to \cdot)$ is a probability distribution over the reductions from $s$, i.e., $\sum_{s\to t} P(s \to t) = 1$; it is assumed that for all $s$ and $t$, $P(s \to t) > 0$ if and only if $s \to t$.*

*The* probability of a finite path $s_0 \to s_1 \to \ldots \to s_n$ *with $n \geq 0$ is given as*

$$P(s_0 \to s_1 \to \ldots \to s_n) = \prod_{i=1}^{n} P(s_{i-1} \to s_i).$$

*For any element $s$ and normal form $t \in R_{NF}(s)$, the* probability of $s$ reaching $t$, *written $P(s \to^* t)$, is defined as*

$$P(s \to^* t) = \sum_{\delta \in \Delta(s,t)} P(\delta);$$

*the* probability of $s$ not reaching a normal form *(or* diverging*) is defined as*

[2] A normalizing system is also called *weakly normalizing* or *weakly terminating* elsewhere, e.g., [34].

$$P(s \to^{\infty}) = 1 - \sum_{t \in R_{NF}(s)} P(s \to^* t).$$

*When referring to the* confluence*,* local confluence*,* termination*, and* normalization *of a PARS, we refer to these properties for the underlying ARS.*

Notice that when $s$ is a normal form, $P(s \to^* s) = 1$ since $\Delta(s,t)$ contains only the empty path with probability $\prod_{i=1}^{0} P(s_{i-1} \to s_i) = 1$. It is important that $P(s \to^* t)$ is defined only when $t$ is a normal form of $s$ since otherwise the defining sum may be $\geq 1$, as demonstrated by the following example.

*Example 7.4.* Consider the PARS $R^P$ given in Figure 7.1a; formally, $R^P = (\{0,1\}, \{0 \to 1, 1 \to 1\}),\ P)$ with $P(0 \to 1) = 1$ and $P(1 \to 1) = 1$. An attempt to define $P(0 \to^* 1)$ as in Def. 7.3, for the reducible element 1, does not lead to a probability, i.e., $P(0 \to^* 1) \not\leq 1$: $P(0 \to^* 1) = P(0 \to 1) + P(0 \to 1 \to 1) + P(0 \to 1 \to 1 \to 1) + \ldots = \infty$.

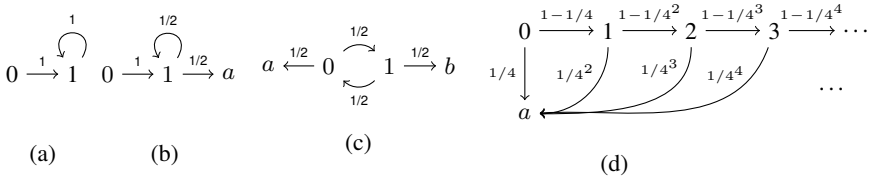



Fig. 7.1: PARS with different properties; see Table 7.1.

The following proposition justifies that we refer to $P$ as a probability function.

**Proposition 7.5.** *For an arbitrary finite path $\pi$, $1 \geq P(\pi) > 0$. For every element $s$, $P(s \to^* \cdot)$ and $P(s \to^{\infty})$ constitute a probability distribution, i.e., $\forall t \in R_{NF}(s)\colon 0 \leq P(s \to^* t) \leq 1$; $0 \leq P(s \to^{\infty}) \leq 1$; and $\sum_{t \in R_{NF}(s)} P(s \to^* t) + P(s \to^{\infty}) = 1$.*

*Proof.* The proofs are simple but lengthy and are given in the Appendix.

Proposition 7.6 justifies that we refer to $P(s \to^{\infty})$ as a probability of divergence.

**Proposition 7.6.** *Consider a PARS that has an element $s$ for which $\Delta^{\infty}(s)$ is countable (finite or infinite). Let $P(s_1 \to s_2 \to \cdots) = \prod_{i=1,2,\ldots} P(s_i \to s_{i+1})$ be the probability of an infinite path; then, $P(s \to^{\infty}) = \sum_{\delta \in \Delta^{\infty}(s)} P(\delta)$ holds.*

*Proof.* See Appendix.

We can now define *probabilistic* and *almost-surely* (abbreviated "a-s.") versions of important concepts for derivation systems. A system is

- *almost-surely convergent* if for all $s_1 \leftarrow^* s \to^* s_2$ there is a normal form $t \in R_{NF}$ such that $s_1 \to^* t \leftarrow^* s_2$ and $P(s_1 \to^* t) = P(s_2 \to^* t) = 1$,

- *locally almost-surely convergent* if for all $s_1 \leftarrow s \rightarrow s_2$ there is a $t \in R_{NF}$ such that $s_1 \rightarrow^* t \leftarrow^* s_2$ and $P(s_1 \rightarrow^* t) = P(s_2 \rightarrow^* t) = 1$,
- *almost-surely terminating*[3] iff every element $s$ has $P(s \rightarrow^\infty) = 0$ and
- *probabilistically normalizing* iff every element $s$ has a normal form $t$ such that $P(s \rightarrow^* t) > 0$.

We have deliberately omitted almost-sure confluence and local confluence [20] since these properties require a more advanced measure to define the probability of visiting a perhaps reducible element.

| | (a) | (b) | (c) | (d) | (d′) |
|---|---|---|---|---|---|
| Loc. confl. | + | + | + | + | + |
| Confl. | + | + | – | + | + |
| Term. | – | – | – | – | – |
| A-s. loc. conv. | – | + | – | – | + |
| A-s. conv. | – | + | – | – | + |
| A-s. term. | – | + | + | – | + |

Table 7.1: A property overview of the systems a–d in Figure 7.1 and (d′) with the same ARS as (d) but with all probabilities replaced by $1/2$.

*Example 7.7.* The four probabilistic systems in Figure 7.1 demonstrate these properties. We notice that b–d are normalizing in $\{a\}$, $\{a, b\}$ and $\{a\}$. Furthermore, they are all non-terminating: system b and c are a-s. terminating, which is not the case for either a or d; for element 0 in system d, we have $P(0 \rightarrow^\infty) = \prod_{i=1}^{\infty}(1 - (1/4)^i) \approx 0.6885 > 0$.[4] Table 7.1 summarizes their properties of (almost-sure) (local) confluence; (d′) refers to a PARS with the same underlying ARS as d and with all probabilities $= 1/2$.

System c is a probabilistic version of a classical example [68, 70], which demonstrates that termination (and not simply a-s. termination) is required for local confluence to imply confluence. The difference between system (d) and system (d′) emphasizes that the choice of probabilities do matter in terms of whether different probabilistic properties hold. For any element $s$ in (d′), the probability of reaching the normal form $a$ is $1/2 + 1/2^2 + 1/2^3 + \cdots = 1$.

## 7.3 Properties of Probabilistic Abstract Reduction Systems

With a focus on almost-sure convergence, we consider now relevant relationships between the properties of probabilistic and their underlying non-probabilistic systems.

[3] Almost-sure termination is named *probabilistic termination* elsewhere, e.g., [56, 130].

[4] Verified by Mathematica. The exact result is $\left(\frac{1}{4}; \frac{1}{4}\right)_\infty$; see [143] for the definition of this notation.

Lemmas 7.9 and 7.11, below, have previously been suggested by [20] without proofs, and we have chosen to include them as well as their proofs to provide a better understanding of the nature of almost-sure convergence. The most important properties are summarized as follows. For any PARS $R^P$:

- $R^P$ is normalizing if and only if it is probabilistically normalizing (Lemma 7.9),
- if $R^P$ is almost-surely terminating, then it is normalizing (Lemma 7.10),
- if $R^P$ is terminating, then it is almost-surely terminating (Lemma 7.11),
- $R^P$ is almost-surely terminating and confluent if and only if it is almost-surely convergent (Theorem 7.12).

The following inductive characterization of the probabilities for reaching a given normal form is useful for the proofs that follow.

**Proposition 7.8.** *For any reducible element $s$, the following holds.*

$$\sum_{t \in R_{NF}} P(s \to^* t) = \sum_{s \to s'} \left( P(s \to s') \times \sum_{t \in R_{NF}} P(s' \to^* t) \right)$$

*Proof.* Any path from $s$ to a normal form $t$ will have the form $s \to s' \to \cdots \to t$ for some direct successor $s'$ of $s$. On the other hand, any normal form for a direct successor $s'$ of $s$ will also be a normal form of $s$. With this observation, the proposition follows directly from Definition 7.3 (prob. of path).

**Lemma 7.9 ([20]).** *A PARS is normalizing if and only if it is probabilistically normalizing.*

*Proof.* Every element $s$ in a normalizing PARS has a normal form $t$ such that $s \to^* t$, and by definition of PARS, $P(s \to^* t) > 0$, which makes it probabilistically normalizing. On the other hand, the definition of probabilistic normalizing includes normalization.

Probabilistic normalization differs from the other properties in nature (requiring probability $> 0$ instead of $= 1$) and is the only property that is equivalent to its non-probabilistic counterpart. Thus, the existing results on proving and disproving normalization can be used directly to determine probabilistic normalization. The following lemma is also a consequence of Proposition 7, parts 3 and 5, of [20].

**Lemma 7.10.** *If a PARS is almost-surely terminating, then it is normalizing.*

*Proof.* For every element $s$ in a almost-surely terminating system, Proposition 7.5 gives that $\sum_{t \in R_{NF}} P(s \to^* t) = 1$, and hence, $s$ has at least one normal form $t$ such that $P(s \to^* t) > 0$. By Lemma 7.9, the system is also normalizing.

The opposite is not the case, as demonstrated by system d in Figure 7.1; every element has a normal form, but the system is not almost-surely terminating.

**Lemma 7.11 ([20]).** *If a PARS is terminating, then it is almost-surely terminating.*

*Proof.* In a terminating PARS, $\Delta^\infty(s) = \emptyset$ for any element $s$. By Proposition 7.6, we have $P(s \to^\infty) = 0$.

The opposite is not the case, as demonstrated by systems b–d in Figure 7.1. The following theorem is a central tool for proving almost-sure convergence.

**Theorem 7.12.** *A PARS is almost-surely terminating and confluent if and only if it is almost-surely convergent.*

Thus, to prove almost-sure convergence of a given PARS, one may use the methods of [19, 50] to prove almost-sure termination and prove classical confluence – referring to Newman's lemma (cf. our discussion in the Introduction) or using the method of mapping the system into another system, already known to be confluent, as described in Section 7.4, below.

*Proof (Theorem 7.12).* We split the proof into smaller parts, referring to properties that are shown below: "if": by Prop. 7.14 and Lemma 7.17. "only if": by Lemma 7.16.

**Lemma 7.13.** *A PARS is almost-surely terminating if it is locally almost-surely convergent.*

*Proof.* Let $R^P$ be a PARS that is locally almost-surely convergent and consider an arbitrary element $s$. We must show $P(s \to^\infty) = 0$ or, equivalently, $\sum_{t \in R_{NF}} P(s \to^* t) = 1$.

When $s$ is a normal form, we have $P(s \to^* s) = 1$ and thus the desired property. Assume, now, that $s$ is not a normal form. This means that $s$ has at least one direct successor; for any two (perhaps identical) direct successors $s', s''$, local almost-sure convergence implies a unique normal form $t_{s',s''}$ of $s'$ as well as of $s''$ with $P(s' \to^* t_{s',s''}) = P(s'' \to^* t_{s',s''}) = 1$. Obviously, this normal form is the same for all such successors and thus a unique normal form of $s$; therefore, let us call it $t_s$. We can now use Proposition 7.8 as follows.

$$\sum_{t \in R_{NF}} P(s \to^* t) = P(s \to^* t_s) = \sum_{s \to s'} \Big( P(s \to s') \cdot P(s' \to^* t_s) \Big) = \sum_{s \to s'} P(s \to s') = 1.$$

This finishes the proof.

Since almost-sure convergence implies local almost-sure convergence, we obtain the weaker version of the above lemma.

**Proposition 7.14.** *A PARS is almost-surely terminating if it is almost-surely convergent.*

The following property for (P)ARS is used in the proof of Lemma 7.16 below.

**Proposition 7.15.** *A normalizing system is confluent if and only if every element has a unique normal form.*

*Proof.* "If": By contradiction: Let $R^P$ be a normalizing (P)ARS; assume that every element has a unique normal form and that $R^P$ is not confluent. By non-confluence, there exist $s_1 \leftarrow^* s \rightarrow^* s_2$ for which there does not exist a $t$ such that $s_1 \rightarrow^* t \leftarrow^* s_2$. However, $s$ has one unique normal form $t'$, i.e., $\{t'\} = R_{NF}(s)$. By the definition of normal forms of $s$, we have that $\forall s' \colon s \rightarrow^* s' \Rightarrow R_{NF}(s) \supseteq R_{NF}(s')$. This holds specifically for $s_1$ and $s_2$, i.e., $\{t'\} = R_{NF}(s) \supseteq R_{NF}(s_1)$ and $\{t'\} = R_{NF}(s) \supseteq R_{NF}(s_2)$. Since $R$ is normalizing, every element has at least one normal form, i.e., $R_{NF}(s_1) \neq \emptyset$ and $R_{NF}(s_2) \neq \emptyset$, leaving one possibility: $R_{NF}(s_1) = R_{NF}(s_2) = \{t'\}$. From this result, we obtain $s \rightarrow^* s_1 \rightarrow^* t'$ and $s \rightarrow^* s_2 \rightarrow^* t'$; this is a contradiction. "Only if": This is a known result; see, e.g., [11].

**Lemma 7.16.** *If a PARS is almost-surely terminating and confluent, then it is almost-surely convergent.*

*Proof.* Lemma 7.10 and Prop. 7.15 ensure that an a-s. terminating system has a unique normal form. A-s. termination also ensures that this unique normal form is reached with probability $= 1$, and thus, the system is almost-surely convergent.

**Lemma 7.17.** *A PARS is confluent if it is almost-surely convergent.*

*Proof.* Assume almost-sure convergence; then, for each $s_1 \leftarrow^* s \rightarrow^* s_2$, there exists a $t$ (a normal form) such that $s_1 \rightarrow^* t \leftarrow^* s_2$.

## 7.4 Showing Probabilistic Confluence by Transformation

The following proposition is a weaker formulation and consequence of Theorem 7.12; it shows that (dis)proving confluence for almost-surely terminating systems is very relevant when (dis)proving almost-sure convergence.

**Proposition 7.18.** *An almost-surely terminating PARS is almost-surely convergent if and only if it is confluent.*

*Proof.* This is a direct consequence of Theorem 7.12 (or using Lemma 7.16 and 7.17).

Curien and Ghelli [34] presented a general method for proving confluence by transforming[5] the system of interest (under some restrictions) to a new system that is known to be confluent. We start by repeating their relevant result.

**Lemma 7.19 ([34]).** *Given two ARS $R = (A, \rightarrow_R)$ and $R' = (A', \rightarrow_{R'})$ and a mapping $G \colon A \rightarrow A'$, $R$ is confluent if the following holds.*

*(C1) $R'$ is confluent,*
*(C2) $R$ is normalizing,*

[5] This is also referred to as interpreting a system elsewhere, e.g., [34].

*(C3) if* $s \to_R t$ *then* $G(s) \leftrightarrow^*_{R'} G(t)$,
*(C4)* $\forall t \in R_{NF},\ G(t) \in R'_{NF}$, *and*
*(C5)* $\forall t, u \in R_{NF},\ G(t) = G(u) \Rightarrow t = u$

We present a version that also permits non-confluence of the transformed system to imply non-confluence of the original system. Notice that (C2)–(C5) is a part of (C2′)–(C5′), and in particular, (C4′) requires additionally that only normal forms are mapped to normal forms.

**Lemma 7.20.** *Given two ARS* $R = (A, \to_R)$ *and* $R' = (A', \to_{R'})$ *and a mapping* $G\colon A \to A'$, *satisfying*

*(C1′) (surjective)* $\forall s' \in A',\ \exists s \in A, G(s) = s'$,
*(C2′)* $R$ *and* $R'$ *are normalizing,*
*(C3′) if* $s \to_R t$ *then* $G(s) \leftrightarrow^*_{R'} G(t)$, *and*
*if* $G(s) \leftrightarrow^*_{R'} G(t)$, *then* $s \leftrightarrow^*_R t$,
*(C4′)* $\forall t \in R_{NF},\ G(t) \in R'_{NF}$, *and* $\forall t' \in R'_{NF},\ G^{-1}(t') \subseteq R_{NF}$,
*(C5′) (injective on normal forms)* $\forall t, u \in R_{NF},\ G(t) = G(u) \Rightarrow t = u$,

*then* $R$ *is confluent iff* $R'$ *is confluent.*

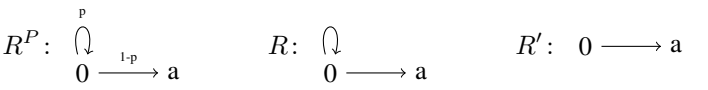
*Proof.* "⇒": follows from Lemma 7.19.
"⇐": Assume that $R$ is confluent and $R'$ is not confluent, i.e., there exist $s'_1 \leftarrow^*_{R'} s' \to^*_{R'} s'_2$ for which $\nexists t' \in R'\colon s'_1 \to^*_{R'} t' \leftarrow^*_{R'} s'_2$.
By (C2′): $\exists t'_1, t'_2 \in R'_{NF}\colon t'_1 \leftarrow^*_{R'} s'_1 \leftarrow^*_{R'} s' \to^*_{R'} s'_2 \to^*_{R'} t'_2$ where $t'_1 \neq t'_2$.
By (C1′) and (C4′): $\exists t_1, t_2 \in R_{NF}\colon G(t_1) = t'_1 \wedge G(t_2) = t'_2$
By (C5′): $t_1 \neq t_2$
By (C3′): $t'_1 \leftrightarrow^*_{R'} t'_2 \Rightarrow t_1 \leftrightarrow^*_R t_2$
By confluence of $R$: $t_1 = t_2$ (contradicts $t_1 \neq t_2$).

We summarize the application of the above to probabilistic systems in Theorems 7.21 and 7.23.

**Theorem 7.21.** *An almost-surely terminating PARS* $R^P = ((A, \to_R), P)$ *is almost-surely convergent if there exists an ARS* $R' = (A', \to_{R'})$ *and a mapping* $G\colon A \to A'$ *that together with* $(A, \to_R)$ *satisfy (C1)–(C5).*

*Proof.* Since $R^P$ is a-s. terminating, $R$ is normalizing (Lemma 7.10). Thus, given an ARS $R'$ and with $G$ as a mapping from $R$ to $R'$ satisfying (C1), (C3)–(C5), we can apply Lemma 7.19 and find that $R$, and thereby, $R^P$ is confluent. A-s. convergence of $R^P$ follows from Prop. 7.18 since $R^P$ is confluent and a-s. terminating

*Example 7.22.* We consider the nonterminating, almost-surely terminating system $R^P$ (below to the left) with the underlying normalizing system $R$ (below, middle), the confluent system $R'$ (below to the right) and the mapping $G(0) = 0$, $G(a) = a$.



The systems $R$, $R'$ and the mapping $G$ satisfy (C1)–(C5), and therefore, we can conclude that $R^P$ is almost-surely convergent.

**Theorem 7.23.** *Given an almost-surely terminating PARS $R^P = (R, P)$ with $R = (A, \to_R)$, an ARS $R' = (A', \to_{R'})$ and a mapping $G$ from $A$ to $A'$ that together with $R$ satisfy (C1′)–(C5′), the system $R^P$ is almost-surely convergent if and only $R'$ is confluent.*

*Proof.* Assume notation as above. Since $R^P$ is a-s. terminating, $R$ is normalizing (Lemma 7.10), thus satisfying the first part of (C2′). Therefore, given an ARS $R'$ and $G$ as a mapping from $A$ to $A'$ that together with $R$ satisfy (C1′)–(C5′), we can apply Lemma 7.19, finding that $R$ is confluent iff $R'$ is confluent. Prop. 7.18 gives that the a-s. terminating $R^P$ is a-s. convergent iff $R'$ is confluent.

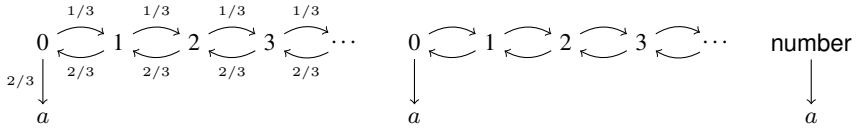
## 7.5 Examples

In the following we show almost-sure convergence in two different cases that exemplifies Theorem 7.23. We use the existing method for showing almost-sure termination [19, 50]: To prove that a PARS $R^P = ((A, \to), P)$ is a-s. terminating, it suffices to show existence of a *Lyapunov ranking function*, i.e., a function $\mathcal{V} : A \to \mathbb{R}^+$ where $\forall s \in A$ there exists an $\epsilon > 0$ so the *inequality of $s$*, $\mathcal{V}(s) \geq \sum_{s \to s'} P(s \to s') \cdot \mathcal{V}(s') + \epsilon$ holds.

### 7.5.1 A Simple, Antisymmetric Random Walk

We consider $R^P = (R, P)$, depicted in Figure 7.2a, a simple positive antisymmetric 1-dimensional random walk. In each step, the value $n$ can either increase to $n + 1$, $P(n \to n + 1) = 1/3$, or decrease to $n - 1$ (or if at 0, we "decrease" to the normal form $a$ instead), $P(n \to n - 1) = P(0 \to a) = 2/3$. Formally, the underlying system $R = (A, \to)$ is defined by $A = \mathbb{N} \uplus \{a\}$ and $\to = \{0 \to a\} \uplus \{n \to n' \mid n, n' \in \mathbb{N}, n' = n + 1 \vee n' = n - 1\}$.
We start by showing $R^P$ a-s. terminating, i.e., that a Lyapunov ranking function exists: let the function $\mathcal{V}$ be defined as follows.

$$\mathcal{V}(s) = \begin{cases} s + 2, & \text{if } s \in \mathbb{N} \\ 1, & \text{if } s = a \end{cases}$$




(a) Original $R^P$. (b) Underlying $R$. (c) $R'$.

Fig. 7.2: Random Walk (1 Dimension)

This function is a Lyapunov ranking since the inequality (see above) holds for all elements $s \in A$; we divide this into three cases, $s > 0$, $s = 0$, and $s = a$:

$$\begin{aligned}
&\mathcal{V}(s) > \tfrac{1}{3} \cdot \mathcal{V}(s+1) + \tfrac{2}{3} \cdot \mathcal{V}(s-1) \Leftrightarrow s + 2 > \tfrac{1}{3}(s+3) + \tfrac{2}{3}(s+1) \; (= s + \tfrac{5}{3}) \\
&\mathcal{V}(0) > \tfrac{1}{3} \cdot \mathcal{V}(1) + \tfrac{2}{3} \cdot \mathcal{V}(a) \qquad \Leftrightarrow 2 > \tfrac{1}{3} \cdot 3 + \tfrac{2}{3} \cdot 1 \text{ , and} \\
&\mathcal{V}(a) > 0 \qquad\qquad\qquad\qquad\quad \Leftrightarrow 1 > 0.
\end{aligned}$$

Since $R^P$ is a-s. terminating, it suffices to define $R' = (\{\mathsf{number}, a\}, \mathsf{number} \to a)$, as seen in Figure 7.2c, and the mapping $G : \mathbb{N} \uplus \{a\} \to \{\mathsf{number}, a\}$.

$$G(s) = \begin{cases} \mathsf{number}, & \text{if } s \in \mathbb{N} \\ a, & \text{otherwise.} \end{cases}$$

Because $R^P$ is a-s. terminating, $R'$ is (trivially) a confluent system, and the mapping $G$ satisfies (C1′)–(C5′); then, $R^P$ is a-s. convergent (by Theorem 7.23).

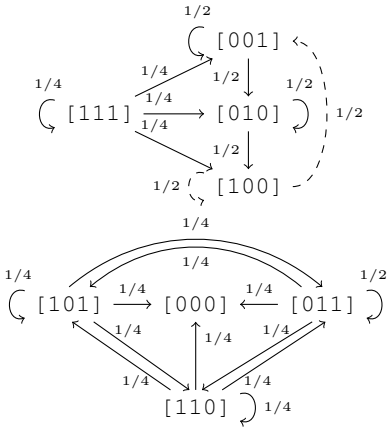
### 7.5.2 Herman's Self-Stabilizing Ring

*Herman's Ring* [67] is an algorithm for self-stabilizing $n$ identical processors connected in an uni-directed ring, indexed 1 to $n$. Each process can hold one or zero tokens, and for each time step, each process either keeps its token or passes it to its left neighbour (-1) with probability $1/2$ of each event. When a process keeps its token and receives another, both tokens are eliminated.

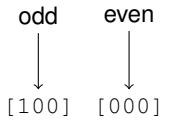
Herman showed that for an initial state with an odd number of tokens, the system will reach a stable state with one token with probability =1. This system is not almost-surely convergent, but proving this for a similar system can be a part of showing that Herman's Ring with 3 processes either will stabilize with 1 token with probability $= 1$ or 0 tokens with probability $= 1$. We use a boolean array to represent whether each process holds a token (`1` indicates a token), and the array is defined as in Figure 7.3, where both dashed and solid edges indicate reductions.
Since `[000]` is a normal form and $\{[\mathtt{100}], [\mathtt{010}], [\mathtt{001}]\}$ is the set of successor states of each of `[100]`,`[010]` and `[001]`, then we can prove the stabilization of $R^P$ by showing almost-sure convergence for a slightly altered system $R'^P$, i.e., the system in Figure 7.3 consisting of the solid edges only.

To show the almost-sure convergence of $R'^P$, we prove almost-sure termination by showing the existence of a Lyapunov ranking function, namely, $\mathcal{V}([b_1\, b_2\, b_3]) = 2^2 \cdot (b_1 + b_2 + b_3) + b_1 \cdot 2^0 + b_2 \cdot 2^1 + b_3 \cdot 2^2$, which decreases, first, with the reduction in tokens and, second, by the position of the tokens. The only two states where $\mathcal{V}$ increases in a direct successor are `[110]` and `[101]`, where the inequality of $[\mathtt{110}]$ reduces to $11 > 9 + \frac{1}{2}$ and that of $[\mathtt{101}]$ to $14 > 9 + \frac{1}{2}$, thereby showing $R^P$ to be a-s. terminating.

We now provide a mapping $G$ from the elements of the underlying system into the elements of a trivially confluent system, i.e., $R''$ in Figure 7.3b:




(a) Original $R^P$ (both dashed and solid edges) and the almost-surely terminating $R'^P$ (without the dashed edges).




(b) Confluent $R''$.

Fig. 7.3: Herman's Self-Stabilizing Ring

$$G([100]) = [100] \qquad G([000]) = [000]$$
$$G([111]) = G([001]) = G([010]) = \mathsf{odd}$$
$$G([011]) = G([101]) = G([110]) = \mathsf{even}$$

The $R^P$ is a-s. terminating, $R''$ is confluent, and $G$ satisfies (C1')–(C5'); then, (by Thm. 7.23) $R^P$ is a-s. convergent.

## 7.6 Related Work

We see our work as a succession of the earlier work by Bournez and Kirchner [20], with explicit and simple definitions (instead of referring to Homogeneous Markov Chain theory) and proofs of central properties. We also show novel properties that are important for showing (non-) convergence. Our work is inspired by the result of [56, 129, 130], given specifically for probabilistic extensions of the programming languages CHR. A concept of so-called nondeterministic PARS has been introduced, e.g., [19, 50], in which the choice of probability distribution for the next reduction is nondeterministic; this is not covered by our results.

PARS can be implemented directly in Sato's PRISM System [121, 122], which is a probabilistic version of Prolog, and recent progress on nonterminating programs [123] may be useful convergence considerations.

## 7.7 Conclusion

We have considered almost-sure convergence – and how to prove it – for probabilistic abstract reduction systems. Our motivation is the application of such systems as computational systems having a deterministic input-output relationship, and therefore, almost-sure termination is of special importance. We have provided properties that are useful when showing almost-sure (non-) convergence by consequence of other probabilistic and "classic" properties and by transformation. We plan to generalize these results to almost-sure convergence modulo equivalence relevant for some Monte-Carlo Algorithms that produce several correct answers (e.g., Simulated Annealing), thereby continuing the work that we have started for (non-probabilistic) CHR [26].

## 7.8 Afterword

In the above, we expand Currien and Ghelli's transformation technique to ensure that confluence or non-confluence is preserved. At least in the examples, the transformed systems can be seen as abstractions of the original systems, and it may be worth investigating whether these abstractions can be automated for certain classes of systems. We suggest using Lyapunov ranking functions for proving almost-sure termination, but there does not yet exist a tool for the automatic detection of such ranking functions. It may be possible to expand the techniques of automatic detection of ordinary ranking functions to the detection of (simple) Lyapunov ranking functions.

Returning to the context of probabilistic analysis, almost-sure convergence implies a functional relation from input to output. In this case there exists a probability measure over output properties, and, therefore, we may use the previously presented approaches.

# 8

# Discussion and Future work

First, we will recall the thesis challenge and survey how we have addressed it. Then, we will discuss future work in the light of three themes.

- Improving precision of the analyses
- Scalability of the proposed methods through compositionality and modularity
- Generalization from deterministic programs to other classes of programs such as probabilistic and nondeterministic programs

The thesis challenge was the following:

> Given a deterministic program, a probability measure over the program inputs and an output/resource property, we want to analyse the probability of the output/resource property. In addition, we want to examine practical approaches to create such probabilistic analyses.

In Chapter 3, we presented an automatic probabilistic output analysis for simple functional programs for the special case of discrete probability distributions and programs over countable input. This approach was the only approach that accepted parametrized input probability distributions as input, and it yielded a mapping from output to an upper probability bound of the output. The analysis was demonstrated by small examples.

We used this approach in a discrete probabilistic resource analysis for C-like programs. We assumed a deterministic and discrete resource model and used that to create a functional program that returned the resource usage of the analysable program. Then, we applied the approach of Chapter 3 to obtain the upper probability bound of the individual resource usages.

In Chapter 5, we took a step back to the general case whereby any input probability measure was allowed, as opposed to only discrete ones. We derived a pair of upper and lower probability bounds for output events, therein using the input probability distribution and a pre-image over-approximating function $pre^\sharp$, *i.e.* a function such that $pre(A) \subseteq pre^\sharp(A)$ whenever $A$ is an output event. We focused on the idea of "reusing existing analyses", either forward or backwards, to obtain the necessary pre-image over-approximating function. We presented two techniques, one for each

case, and exemplified the forward case by minor examples produced systematically using implemented analysis and calculating the probability bounds by hand.

In Chapters 6 and 7, we presented approaches to nondeterministic and probabilistic programs that determined whether the program semantics could be seen as a function that in return ensured that there exist a probabilistic output measure. This is a step towards applying the techniques in Chapter 5 to probabilistic and nondeterministic programs.

For nondeterministic programs, we identified the property confluence modulo equivalence that ensures that the program's input-output relation can be seen as a function from input to classes of equivalent outputs, which in return ensures that there exists an output probability measure over these classes[1]. Note that when analysing a resource-instrumented program, confluence modulo equivalence must hold for the resources.

For probabilistic programs, we identified the property *almost-surely convergent*, that is, for each input, the program reaches a unique output element with probability 1. We showed that a program is almost-surely convergent if and only if it is almost-surely terminating, that is, it terminates with probability 1, and is confluent; there exist methods for proving both of these properties [19, 34, 50, 132].

## 8.1 Precision

In Chapter 3, we developed a probabilistic analysis for programs over countable input; they lie within the general framework but are amenable to analytical solutions. This approach approximates solutions by solving recurrence equations and uses the computer algebra system Mathematica [146] to handle summations and products symbolically. The approach computes upper probability bounds. The analyses have been tested on a series of small examples, and the approach derived precise bounds for the tested programs. However, when the recurrence equations are not solvable, the method defaults to the trivial bounds. In automatic complexity analysis, the recurrence equation solutions are approximated [7, 127], and it derives a pair of upper and lower bounds instead of a single upper bound. Such an approach could improve the analyses. For instance, let `f` be a function defined as follows, where `(*)` indicates some boolean test whereby we do not know whether it evaluates to true or false. We assume that we call `f`, and `i` equals 1.

```
f(i,y) = if i>=2 then y else f(i+1, g(y))
g(y)   = if (*) then y+1 else y
```

Let the probability distribution over `f`'s input describe the fact that there is an equal chance that `y` has the value 0 and 1. The output of `f(1,0)` is either 0 or 1, and the output of `f(1,1)` is either 1 or 2. Then, the upper probability bound $P_{\mathtt{f}}$ of `f`'s output distribution is $P_{\mathtt{f}}(z) = \sum_{x\in\{0,1\}} 1/2 \cdot c(z = \mathtt{f}(1,y))$, where $c(\textit{some test})$

[1] To be precise, there exists an output probability measure over a $\sigma$-algebra containing the emptyset, these classes and all possible unions of these classes.

returns 1 if *some test* evaluates to true and 0 otherwise. If we assume that we have expanded the analysis such that it can approximate the recurrence equations of `f` such that `f(1,`$y$`)` may produce either $y$ or $y+1$, we could achieve the following upper bound for the output distribution.

$$\begin{aligned} P_{\mathtt{f}}(z) &= \sum_{x\in\{0,1\}} 1/2 \cdot c(z = \mathtt{f}(1,x)) \\ &= 1/2 \cdot c(z = \mathtt{f}(1,0)) + 1/2 \cdot c(z = \mathtt{f}(1,1)) \\ &= 1/2 \cdot c(0 \leq z \leq 1) + 1/2 \cdot c(1 \leq z \leq 2) \end{aligned}$$

This is better than the currently possible $P_{\mathtt{f}}(z) = 1$, *i.e.* better than the result given by the method in Chapter 3.

A second point reveals itself if we analyse the probability bound of an output event consisting of at least two outputs, for instance, the output event $\{0,1\}$; by definition of $P_{\mathtt{f}}$, $P_{\mathtt{f}}(0) + P_{\mathtt{f}}(1) = 1\frac{1}{2}$ is such an upper bound. However, if we consider the computed $P_{\mathtt{f}}$ using the knowledge from Chapter 5, we see that the $c$-inequalities in $P_{\mathtt{f}}$ express $img^\sharp(\mathtt{f}(1,0)) = \{0,1\}$ (the $c(0 \leq z \leq 1)$) and $img^\sharp(\mathtt{f}(1,1)) = \{0,1\}$ (the $c(1 \leq z \leq 2)$). The $1/2$s are their respective input probabilities. From Theorem 5.21, we know that these fragments together express not only the probability of single outputs but also the boundaries of sets of output. Specifically, $P_{\mathtt{f}}$ describes the upper and lower probabilities of output events. For instance, the upper and lower probabilities of the output event $\{0,1\}$ are 1 and $1/2$, respectively; the upper bound is 1 since $\{0,1\}$ overlaps with both $img^\sharp(\mathtt{f}(1,0))$ and $img^\sharp(\mathtt{f}(1,1))$, and the lower bound is $1/2$ because $\{0,1\}$ is a subset of $img^\sharp(\mathtt{f}(1,0)$ but not of $img^\sharp(\mathtt{f}(1,1)$.

Another aspect of precision is related to the approach in Chapter 5. Here, choosing a more precise "black-box" analysis than those used in the experimental results may yield more precise results. We could for instance use a convex polyhedron analysis [32] that obtains better precision than the interval analysis. Other analyses combine different techniques, *e.g.* [74], to obtain even more precise results. If these better but more complex analysis results are not measurable in the input measure, they must be abstracted into measurable sets, and the choice of abstraction affects the precision (Section 5.2.1). For instance, let us say that we have a program that takes two independent variables $a, b$ as input and that their probability measure would be defined over all rectangles $[(x_a, y_a), (x_b, y_b)]$. On the other hand, let us say that the analysis result given some output event is a polyhedron $a + b < 5,\ 1 \leq a \leq 2,\ 1 \leq b \leq 2$, as indicated by the solid lines in Figure 8.1. This result is not measurable, and we need to choose some enclosing rectangles. For instance, $[(1,1),(2,2)]$ and $[(1,1),(2,1.5)] \uplus [(1,1.5),(1.5,2)]$ are both measurable over-approximations of the polyhedron (see Figure 8.1), but the latter is more precise and may lead to a more precise probability bound.


Fig. 8.1: If the input event is not measurable, it must be abstracted to a measurable input. The choice affects the precision of the probability bound of the output event.

## 8.2 Scaling up

The results presented in this thesis are mainly theoretical and are illustrated with small examples produced either by prototypes or systematically by hand. All the analyses are developed as whole-program analyses and as such are unlikely to scale. In the following, we will discuss two approaches to the probabilistic analysis of a large software system, $sys$, consisting of many components $f_1, \ldots, f_n$.

Approach 1. Apply non-probabilistic analysis to each component $f_i$ separately, acquiring pre-image approximations $pre^\sharp_{f_i}$; obtain $pre^\sharp_{sys} = F_{pre}(pre^\sharp_{f_1}, \ldots, pre^\sharp_{f_n})$ by their composition; and apply the techniques in Chapter 5 or similarly for $img^\sharp_{f_i}$.

Approach 2. Apply probabilistic analysis to each component $f_i$ separately, acquiring a transformer $P_{f_i}$ from probability distributions to sets of probability distributions, *e.g.*, as in Chapter 3-4 or [33, 95], and obtain $P_{sys} = F_P(P_{f_1}, \ldots, P_{f_n})$, which maps the input distribution for the whole system to a set of output distributions.

One of the main challenges with both approaches is that the components are not completely independent[30], *e.g.*, the variables in the state may depend on each other.

*Approach 1.* Consider an interval analysis of the following program `f`.

```
f(x) = g(x,h(x))
g(x,y) = x-y
h(x) = x
```

Interval analysis[2] of `g` with independent arguments is precise, *i.e.*, `g([w, x],[y, z]) = [w - z, x - y]`, as is that of `h`, *i.e.*, `h([w, x]) = [w, x]`. However, the analysis of `f` is imprecise due to the repeated argument `x`, *i.e.*, $img^\sharp_f$ is imprecise due to the dependence between the arguments of `g`. The image approximation yields $img^\sharp_f([w,x]) = img^\sharp_g([w,x],[w,x]) = [w-z, z-x]$, where for instance

[2] Moore described interval arithmetic to evaluate the ranges of functions taking interval arguments; one rule was the interval minus operator $[w,x] - [y,z] = [w-z, x-y]$ [100].

$img_f^\sharp([0,1]) = img_g^\sharp([0,1], img_h^\sharp([0,1])) = img_g^\sharp([0,1],[0,1]) = [0-1, 1-0] = [-1,1]$, whereas the precise answer is $[0,0]$.

One solution to Approach 1 is to find more precise domains that can capture dependencies, *E.g.*, a relational domain, such as convex polyhedra, gives a precise solution for `f`. However, relational abstract interpretations are more expensive to compute. Other solutions, such as creating dependency graphs and analysing dependent components together, are mentioned in [30]. However, that may be equivalent to a whole system analysis in the extreme case whereby all components are dependent.

*Approach 2.* Consider the same program as before (program `f`). Let $P_g$ and $P_h$ be probability distribution mappings for `g` and `h`, respectively. We wish to construct a $P_f$ that maps an input distribution to an output distribution. Given a distribution $\mu_X$ over input $X$, the distributions $\mu_X$ and $P_h(\mu_X)$ are obviously dependent. As demonstrated by Example 8.1, it is crucial to consider this dependency. To obtain an accurate result, we should apply $P_g$ to a joint probability distribution, thereby capturing the dependency.

*Example 8.1.* Let `f` be the program from above, and let `x` be a uniform distribution over $[0,1]$. The input values of the call to `g` are dependent in that they have the same values. Their shared probability distribution is depicted in Figure 8.2a. The correct result of `g(x,h(x))` is depicted in Figure 8.2b by orange intervals (with length 0.2) together with the result of `g(x,h(x))`, where `x` and `h(x)` are (incorrectly) assumed independent (black 0.2-intervals). The independence result is neither an upper nor lower probability bound of `f`'s output.

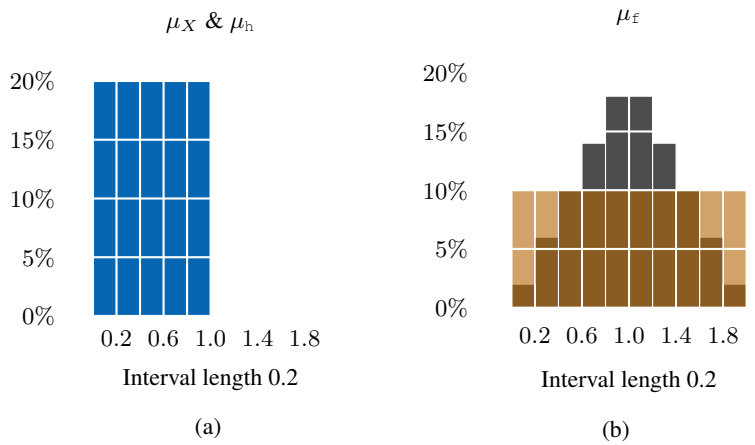



Fig. 8.2: (a) The input probability of `x` and output probability of `h`. (b) The correct and incorrect output probability of `f` when the input arguments are (incorrectly) assumed independent (black) and when they are assumed fully dependent (orange).

Copulas provide a framework for constructing a joint distribution as a function of two independent probability distributions. The problem then is to construct appropriate copulas to represent the dependencies that arise (*e.g.*, the dependency between the distributions $\mu_X$ and $P_h(\mu_X)$ in the example). In general, we may not be able to determine independence nor dependence between a pair of distributions; however, the possible copulas are encapsulated by the Fréchet-Hoeffding bounds, *e.g.* [104], and thus, the set of possible joint probability distributions are restricted. In future work, we might draw on previous studies of these bounds when defined on (different representations of) sets of probability measures, *e.g.* [16, 47, 98, 99].

## 8.3 Generalizing to other classes of programs

*Non-deterministic Programs* For the approaches in Chapter 3 and Chapter 4, the analysis will have to be improved; the biggest challenge is that they cannot produce nor handle approximations of recurrence relations, that is, recurrence inequations [7, 127], which is necessary for non-deterministic programs.

In retrospect, the techniques presented in Chapter 5 could have been expanded to non-deterministic programs. A non-deterministic program relation $|\mathtt{prg}|$ is a total one-to-many relation from input to outputs; the pre-image $pre_{|\mathtt{prg}|}$, dual pre-image $\widetilde{pre}_{|\mathtt{prg}|}$, and images $img_{|\mathtt{prg}|}$ are well defined, but here, $\widetilde{pre}_{|\mathtt{prg}|} \subseteq pre_{|\mathtt{prg}|}$. Naturally, neither $\mu \circ pre_{|\mathtt{prg}|}$ nor $\mu \circ \widetilde{pre}_{|\mathtt{prg}|}$ infer probability measures in general; instead, they infer upper and lower probability bounds [23, 106]. The functions $\mu \circ pre_{|\mathtt{prg}|}$ and $\mu \circ \widetilde{pre}_{|\mathtt{prg}|}$ are known as a plausibility $pl$ and belief $bel$ function [147], respectively, and both have properties closely related to probability measures. Their nature is such that $Pl \circ pre_{|\mathtt{prg}|}$ is also a plausibility function, and $Bel \circ pre_{|\mathtt{prg}|}$ is a belief function. When $Pl$ and $Bel$ are dual, $Pl \circ pre_{|\mathtt{prg}|}$ and $Bel \circ pre_{|\mathtt{prg}|}$ are dual as well, *i.e.*, $Pl(pre_{|\mathtt{prg}|}(A)) = 1 - (Bel(pre_{|\mathtt{prg}|}(A^{\complement}))$.

Adapting the main theorem in Chapter 5 (Theorem 5.10) to multivalued mappings requires proving $pre^{\flat}_f \subseteq \widetilde{pre}_f \subseteq pre_f \subseteq pre^{\sharp}_f$ when assuming $pre_f \subseteq pre^{\sharp}_f$; this is a consequence of $\widetilde{pre}_f \subseteq pre_f$ and $pre^{\sharp}_f(A^{\complement}) \supseteq pre_f(A^{\complement}) \Leftrightarrow pre^{\sharp}_f(A^{\complement})^{\complement} \subseteq pre_f(A^{\complement})^{\complement} \Leftrightarrow pre^{\flat}_f(A) \supseteq \widetilde{pre}_f(A)$. The proof of duality needs no changes.

*Probabilistic Programs* As we saw in the experimental results of Chapter 5, in trivial cases, we may be able to transform probabilistic programs into non-probabilistic programs, where the random generators are given as input. Speculatively, such a transformation would require that the number of calls to random generators be bounded by a fixed number and that there exists a projector function such that the execution of a random generator call in the original program can be exchanged with a unique input (also for calls occurring in loop bodies). It might not be possible to define such bounds precisely, but the methods of loop bound analysis may provide us with an over-approximation. When the random generators are independent of each other, their input must be independent, and when they are dependent, this dependency should be reflected as well.

The theory in Chapter 5 cannot handle probabilistic programs; however, provided an expansion to the handling of non-deterministic programs, it may provide crude upper and lower bounds for programs where we map random generators to intervals $[0, 1]$ (with probability 1).

Another possibility could be to mimic the method used by Monniaux in his output analysis, simply partition the output of each random generator, and tag them with their probability/weight. This requires an execution to maintain a list of possible interval and weight pairs. One advantage would be that it may be more precise than the crude abstraction mentioned above, but a disadvantage is that it produces a combinatorial explosion each time a new random generator call is encountered in an execution. Adje *et al*. [4] had a similar issue and suggested limiting the number of elements and combining the intervals and weights dynamically when the limit was exceeded, *e.g*., the list $\{([0, 1], 0.2), ([0, 2], 0.2)\}$ may be abstracted/reduced to $\{([0, 2], 0.4)\}$.

## 8.4 Related work

The work presented in the Chapters 3, 4 builds on area of resource analysis [8, 9, 15, 86–88, 103, 109, 127] leading all the way back to Wegbreit's essential article in 1975 [141]. The transformational approach was introduced by Burstall and Darlington [22] and formed the basis of the transformational approaches within automatic complexity analysis with the essential work by Le Metayer [91] and by Rosendahl [116, 117]. Essentially these works follow the same procedure: first derive recurrence relations (or cost relations) and second solve those to obtain polynomial, logarithmic, or exponential resource bounds; the advantage being that the bounds are not constrained to linear bounds.

We extend this technique to handle probabilistic resource analysis (Chapters 3 and 4). The main novelty being that we express and solve probabilistic equations typically involving sum and product expressions. Both approaches have been illustrated on simple examples, yet it remains to be seen whether one may achieve the expected precision for programs with more complex dataflow or whether they merely produce the trivial upper bound 1 (for each output).

The approaches presented in Chapter 5 builds on abstract interpretation introduced by Cousot and Cousot in 1977 [28, 29]; the classical abstract domains being intervals [27–29] and polyhedra [32, 64, 65], but more elaborate domains are developed more recently, for instance octagons [92–94] or ellipsoids [107]. Abstract interpretation is mainly used when analysing deterministic and non-deterministic programs and to a minor degree when analysing probabilistic programs [4, 39, 97]. The two essential works by Monniaux [95, 96] and by Cousot and Monerau [33] each describe a framework for extending existing non-probabilistic analysis (described using abstract interpretation) to probabilistic and nondeterministic program analysis.

The work presented in Chapter 5, relates to these frameworks. While both frameworks can analyse a larger class of programs, namely probabilistic programs, they

require manual development of a new semantics that handles random generators. One advantage of the presented methods of Chapter 5 is that the original analysis may be seen as black box, and that the result of several black-box analyses may be combined in order to get a better approximation of the input leading to a given property, hence providing tighter bounds. This makes the presented approaches more amenable to implementation, and future work includes constructing such an implementation based on existing tools.

As indicated by the experiments of Chapter 5, using Monniaux's framework to lift an analysis produces a less or equally precise probabilistic analysis than the one obtained by applying the methods presented in Chapter 5 to the same analysis using the same abstraction. The imprecision is caused by the manner in which their probabilistic abstract semantics propagate probabilities through if-statements. When deducing the upper probability bound of an output event $B$, it is essential that the input-probability $\mu$ of each partition element $t$ may contribute to $B$'s probability once.

*Example 8.2.* In this example we compare Monniaux's probabilistic interval analysis and the forward approach where we fix the abstract domains so they correspond in coarseness. We analyse the following program

```
f(x,y) = if (x>2) then y += 1 else y += 2; (*) return y;
```

where the (concrete) input probability measure describes that there is an equal chance that x and y are (independently) uniformly distributed in the interval $[1, 3]$ and that they are (independently) uniformly distributed in the interval $[3, 5]$.

In the following we analyse the output event $[3, 4]$ for program f using first Monniaux's framework and then using the forward approach. For comparison we have manually calculated the correct probability of the output event, namely $0.25$. [3] We fix the abstract domain/input partition so we obtain to two states (i) where x maps to $[1, 3]$ and y maps to $[1, 3]$ and (ii) where x maps to $[3, 5]$ and y maps to $[3, 5]$. Because the value of x variable has no influence on the output after the branching condition, we will exclude it from the states for simplicity.

The interval analysis derives that for state (i) where x and y maps to $[1, 3]$ the "then" branch yields y maps to $[2, 4]$ and the "else" branch yields y maps to $[3, 5]$, thus, they are both possible environments at program point (*). For state (ii) where x and y maps to $[3, 5]$ only the "then" branch is feasible and yields that y maps to $[4, 6]$.

Monniaux's probabilistic interval analysis: for case (i) x and y maps to $[1, 3]$ with probability $0.5$, the "then" branch yields $[2, 4]\star 0.5$ meaning that y may be described by any measure $\mu$ such that $\mu([2, 4]) = 0.5$ and with total weight $0.5$. The "else" branch yields $[3, 5]\star 0.5$. For case (ii) where x and y maps to $[3, 5]$ with probability $0.5$, the "then" branch yields $[4, 6]\star 0.5$. After the if-statement, at program point (*), the states are summed, obtaining $[2, 4]\star 0.5 + [3, 5]\star 0.5 + [4, 6]\star 0.5$ meaning y may

[3] The total of $0.25$ stems from two parts of the input: $0.125$ from the input cases $x \in [2, 3]$ and $y \in [2, 3]$ where y is incremented by 1 in the "then" branch, and another $0.125$ from the input cases $x \in [1, 2)$ and $y \in [1, 2]$ where y is incremented by 2 in the "else" branch.

be described by any measure $\mu$ which is the sum[4] of three measures $\mu = \mu' + \mu'' + \mu'''$ such that $\mu'([2,4]) = 0.5$, $\mu''([3,5]) = 0.5$, $\mu'''([4,6]) = 0.5$, and each has a total weight of $0.5$. The analysis obtain the upper probabilistic bound $1$ for the output event $[3,4]$.

The forward approach of Chapter 5 yields that for case (i) either $[2,4]$ or $[3,5]$ is reached, and for case (ii) only $[4,6]$ is reached, thus only the partition relating to the output event where $\mathtt{y}$ is a value in $[3,4]$ is case (i). The approach provides an upper probability bound of $0.5$ for the output event $[3,4]$.

We conclude that in the above example[5] the forward approach gave a tighter upper bound than the analysis created using Monniaux's framework. However, it remains to be shown formally.

[4] The sum of two measures $\mu = \mu' + \mu''$ (over the same sigma-algebra) is defined as $\mu(A) = \mu'(A) + \mu''(A)$.

[5] Another example is program $\mathtt{h}$ on page 77 with results presented in Figure 5.6.

# A

# Relations, functions and multi-valued mappings.

**Definition A.1.** *The* cartesian product *of the sets $X$ and $Y$ is the set of all pairs $(x, y)$ such that $x \in X$ and $y \in Y$,* i.e. $X \times Y = \{(x, y) \mid x \in X, y \in Y\}$. *We write $X \times Y \times Z$ for $(X \times Y) \times Z$ and $X^n$ for $\underbrace{X \times \ldots \times X}_{n \text{ times}}$.*

**Definition A.2.** *A* relation *is any subset of a cartesian product. An $n$-*ary relation *is a set of $n$-tuples. An $n$-ary relation $R$ is a* relation on *$X$ if $R \subseteq X^n$.*

**Definition A.3.** *If $R$ is a binary relation, the* domain *of $R$ is* $\mathrm{dom}(R) \triangleq \{x \mid \exists y\colon (x, y) \in R\}$,*and the* range *of $R$ is* $\mathrm{ran}(R) \triangleq \{y \mid \exists x\colon (x, y) \in R\}$.

**Definition A.4.** *A relation $R \subseteq X \times Y$ is* total *if* $\mathrm{dom}(R) = X$ *and* partial *otherwise.*

**Definition A.5.** *A binary relation $f \subseteq X \times Y$ is a* function *if $(x, y) \in f \wedge (x, z) \in f \Rightarrow y = z$. The* value *of $f$ at $x$ is $y$ if $(x, y) \in f$; we use the functional notation $y = f(x)$. $f$ is a* function on *$X$ if* $\mathrm{dom}(f) = X$. *If* $\mathrm{dom}(F) = X^n$, *then $f$ is an $n$-*ary function on *$X$. $f$* is a function from *$X$* to *$Y$, $f\colon X \to Y$, if* $\mathrm{dom}(f) = X$ *and* $\mathrm{ran}(f) \subseteq Y$.

Note that the notation $f\colon X \to Y$ and saying "$f$ is a function on $X$" implicitly provide that $f$ is a total function.

**Definition A.6.** *A relation $R \subseteq X \times Y$* over-approximates *a relation $R' \subseteq X \times Y$ if $R' \subseteq R$.*

**Lemma A.7.** *Let a relation $R \subseteq X \times Y$ be total, and let $R' \subseteq X \times Y$ over-approximate $R_f$,* i.e.*, $R \subseteq R'$; then, $R'$ is also total.*

*Proof.* For all $x \in X$, there is a pair $(x, y)$ in $R$ because $R$ is total. Since $R \subseteq R'$, we find that, for all $x \in X$, there is a pair $(x, y)$ in $R'$. Thus, $R'$ is total.

# B

# Selected proofs for Chapter 7

**Proposition B.1.** *For an arbitrary finite path $\pi$, $1 \geq P(\pi) > 0$. For every element $s$, $P(s \to^* \cdot)$ and $P(s \to^\infty)$ constitute a probability distribution, i.e., $\forall t \in R_{NF}(s)\colon 0 \leq P(s \to^* t) \leq 1$; $0 \leq P(s \to^\infty) \leq 1$; and $\sum_{t \in R_{NF}(s)} P(s \to^* t) + P(s \to^\infty) = 1$.*

*Proof.* Part one follows by Definition 7.3. Part two is shown by defining a sequence of distributions $P^{(n)}$, $n \in \mathbb{N}$, only containing paths up to length $n$, and we show that it converges to $P$. Let $\Delta^{(n)}(s,t)$ be the subset of $\Delta(s,t)$ with paths of length $n$ or less, and let $\Delta^{(n)}(s,\sharp)$ be the set of paths of length $n$, starting in $s$ and ending in a reducible element.
We can now define $P^{(n)}$ over $\{\Delta^{(n)}(s,t) \mid t \in R_{NF}(s)\} \uplus \{\Delta^{(n)}(s,\sharp)\}$ as follows:

$$P^{(n)}(s \to^* t) = \textstyle\sum_{\delta \in \Delta^{(n)}(s,t)} P(\delta), \quad \text{and} \tag{B.1}$$

$$P^{(n)}(s \to^\infty) = \textstyle\sum_{\pi \in \Delta^{(n)}(s,\sharp)} P(\pi). \tag{B.2}$$

First, we prove by induction that $P^{(n)}$ is a distribution for all $n$. The $P^{(0)}$ is a distribution because (i) If $s$ is irreducible, $P^{(0)}(s \to^* s) = 1$ (the empty-path); $P^{(0)}(s \to^\infty) = 0$ (a sum of zero elements). (ii) If $s$ is reducible, $P^{(0)}(s \to^* s) = 0$; and $P^{(0)}(s \to^\infty) = \sum_{s \to t} P(s \to t) = 1$ by Definition 7.3.
The inductive step: The sets $\Delta^{(n+1)}(s,t)$, $t \in R_{NF}(s)$, and $\Delta^{(n+1)}(s,\sharp)$ can be constructed by, for each path in $\Delta^{(n)}(s,\sharp)$, creating its possible extensions by one reduction. When an extension leads to a normal form $t$, it is added to $\Delta^{(n)}(s,t)$; otherwise, i.e., if the new path leads to a reducible, it is included in $\Delta^{(n+1)}(s,\sharp)$. Formally, for any normal form $t$ of $s$, we write

$$\Delta^{(n+1)}(s,t) = \{(s \to \cdots \to u \to t) \mid (s \to \cdots \to u) \in \Delta^{(n)}(s,\sharp),\ u \to t\} \uplus \Delta^{(n)}(s,t)$$
$$\Delta^{(n+1)}(s,\sharp) = \{(s \to \cdots \to u \to v) \mid (s \to \cdots \to u) \in \Delta^{(n)}(s,\sharp),\ u \to v,\ u \notin R_{NF}(s)\}$$

We show that for a given $s$, the probability mass added to the $\Delta^{(\cdot)}(s,t)$ sets is equal to the probability mass removed from $\Delta^{(\cdot)}(s,\sharp)$ as follows (where $\delta_{su} = (s \to \cdots \to u)$).

$$\sum_{t\in R_{NF}(s)} P^{(n+1)}(s\to^* t) + P^{(n+1)}(s\to^\infty) = \sum_{\substack{t\in R_{NF}(s)\\ \delta\in\Delta^{(n+1)}(s,t)}} P^{(n+1)}(\delta) + P^{(n+1)}(s\to^\infty)$$

$$= \sum_{\substack{t\in R_{NF}(s)\\ \delta_{st}\in\Delta^{(n)}(s,t)}} P^{(n)}(\delta) \;+\; \sum_{\substack{\delta_{su}\in\Delta^{(n)}(s,\sharp),\\ u\to v, v\in R_{NF}(s)}} P^{(n)}(\delta)P(u\to v) \;+\; \sum_{\substack{\delta_{su}\in\Delta^{(n)}(s,\sharp),\\ u\to v, v\notin R_{NF}(s)}} P^{(n)}(\delta)P(u\to v)$$

$$= \sum_{t\in R_{NF}(s)} P^{(n)}(s\to^* t) + \sum_{\substack{\delta_{su}\in\Delta^{(n)}(s,\sharp),\\ u\to v}} P^{(n)}(\delta)P(u\to v) = \sum_{t\in R_{NF}(s)} P^{(n)}(s\to^* t) + \sum_{\delta_{su}\in\Delta^{(n)}(s,\sharp)} P^{(n)}(\delta)\left(\sum_{u\to v} P(u\to v)\right)$$

$$= \sum_{t\in R_{NF}(s)} P^{(n)}(s\to^* t) + P^{(n)}(s\to^\infty) = 1$$

Thus, for given $s$, $P^{(n+1)}$ defines a probability distribution. Notice also that the equations above indicate that $P^{(n+1)}(s\to^* t) \geq P^{(n)}(s\to^* t)$ for all $t \in R_{NF}(s)$.

Finally, for any $s$ and $t \in R_{NF}(s)$, $\lim_{n\to\infty}\Delta^{(n)}(s,t) = \Delta(s,t)$, we obtain (as we consider increasing sequences of real numbers in a closed interval) $\lim_{n\to\infty} P^{(n)}(s\to^* t) = P(s\to^* t)$, and consequently, $\lim_{n\to\infty} P^{(n)}(s\to^\infty) = P(s\to^\infty)$. This finishes the proof.

**Proposition B.2.** *Consider a PARS that has an element $s$ for which $\Delta^\infty(s)$ is countable (finite or infinite). Let $P(s_1\to s_2\to\cdots) = \prod_{i=1,2,\ldots} P(s_i\to s_{i+1})$ be the probability of an infinite path; then, $P(s\to^\infty) = \sum_{\delta\in\Delta^\infty(s)} P(\delta)$ holds.*

*Proof.* We assume the characterization in the proof of Proposition 7.5 above and of $P$ by the limits of the functions $P^{(n)}(s\to^* t)$ and $P^{(n)}(s\to^\infty)$ given by equations (B.1) and (B.2). When $\Delta^\infty(s)$ is countable, $\lim_{n\to\infty} P^{(n)}(s\to^\infty) = \sum_{\delta\in\Delta^\infty(s)} P(\delta)$.

# References


[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. G. Smolka, redaktor, *CP, Constraint Programming*, wolumen 1330 serii *LNCS*, strony 252–266. Springer, 1997.

[2] S. Abdennadher, T. Frühwirth, H. Meuss. Confluence and Semantics of Constraint Simplification Rules. *Constraints*, 4(2):133–165, 1999.

[3] S. Abdennadher, T. W. Frühwirth, H. Meuss. On confluence of constraint handling rules. *CP96*, wolumen 1118 serii *LNCS*, strony 1–15. Springer, 1996.

[4] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault, S. Putot. *Static Analysis of Programs with Imprecise Probabilistic Inputs*, strony 22–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[5] A. V. Aho, R. Sethi, J. D. Ullman. Code Optimization and Finite Church-Rosser Systems. R. Rustin, redaktor, *Design and Optimization of Compilers*, strony 89–106. Prentice-Hall, 1972.

[6] E. Albert, P. Arenas, S. Genaim, G. Puebla. Cost analysis of java bytecode. *Languages and Systems*, 2007.

[7] E. Albert, P. Arenas, S. Genaim, G. Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, 5079:221–237, 2008.

[8] E. Albert, P. Arenas, S. Genaim, G. Puebla. Cost relation systems: A language-independent target language for cost analysis. *Electronic Notes in Theoretical Computer Science*, 248(Supplement C):31 – 46, 2009. Proceedings of the Eighth Spanish Conference on Programming and Computer Languages (PROLE 2008).

[9] E. Albert, P. Arenas, S. Genaim, G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, Feb 2011.

[10] T. Augustin, F. P. A. Coolen, G. de Cooman, M. C. M. Troffaes, redaktorzy. *Introduction to Imprecise Probabilities*. Wiley Series in Probability and Statistics. John Wiley & Sons, Ltd, Chichester, UK, may 2014.

[11] F. Baader, T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.

[12] L. Babai. Monte-Carlo algorithms in graph isomorphism testing. *Université de Montréal Technical Report, DMS*, (79):1–33, 1979.

[13] C. Baier, J.-P. Katoen. *Principles Of Model Checking*, wolumen 950. 2008.

[14] M. Bauer. Approximations for Decision Making in the Dempster-Shafer Theory of Evidence. E. Horvitz, F. V. Jensen, redaktorzy, *UAI*, strony 73–80. Morgan Kaufmann, 1996.

[15] R. Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*, 318(1-2):79–103, 2004.

[16] D. Berleant, H. Cheng. A Software Tool for Automatically Verified Operations on Intervals and Probability Distributions. *Reliable Computing*, 4(1):71–82, 1998.

[17] G. Bernat, A. Burns, M. Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Computing*, 1(2):179–194, 2005.

[18] O. Bouissou, E. Goubault, J. Goubault-Larrecq, S. Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 94(2-4):189–201, 2012.

[19] O. Bournez, F. Garnier. Proving positive almost sure termination under strategies. F. Pfenning, redaktor, *RTA 2006*, wolumen 4098 serii *LNCS*, strony 357–371. Springer, 2006.

[20] O. Bournez, C. Kirchner. Probabilistic rewrite strategies. Applications to ELAN. S. Tison, redaktor, *RTA 2002*, wolumen 2378 serii *LNCS*, strony 252–266. Springer, 2002.

[21] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-Garcia, G. Puebla. The Ciao prolog system. *Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM)*, 95:96, 1997.

[22] R. Burstall, J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.

[23] G. Choquet. Theory of capacities. *Annales de l'institut Fourier*, 5:131–295, 1954.

[24] H. Christiansen, C. T. Have, O. T. Lassen, M. Petit. The Viterbi Algorithm expressed in Constraint Handling Rules. P. Van Weert, L. De Koninck, redaktorzy, *Proceedings of the 7th International Workshop on Constraint Handling Rules*, Report CW 588, strony 17–24. Katholieke Universiteit Leuven, Belgium, 2010.

[25] H. Christiansen, M. H. Kirkeby. Confluence Modulo Equivalence in Constraint Handling Rules. wolumen 8981, strony 41–58. Springer International Publishing Switzerland, 2015.

[26] H. Christiansen, M. H. Kirkeby. On proving confluence modulo equivalence for Constraint Handling Rules. *Formal Aspects of Computing*, 29(1):57–95, jan 2017.

[27] P. Cousot, R. Cousot. Static determination of dynamic properties of programs. *Proceedings of the Second International Symposium on Programming*, strony 106–130. Dunod, Paris, France, 1976.

[28] P. Cousot, R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, 1977.

[29] P. Cousot, R. Cousot. Systematic design of program analysis frameworks, 1979.

[30] P. Cousot, R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. *Proceedings of the Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet, SSGRR*, strony 6–12, 2001.

[31] P. Cousot, R. Cousot, M. Fähndrich, F. Logozzo. Automatic inference of necessary preconditions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7737 LNCS:128–148, 2013.

[32] P. Cousot, N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. A. V. Aho, S. N. Zilles, T. G. Szymanski, redaktorzy, *POPL*, strony 84–96. ACM Press, 1978.

[33] P. Cousot, M. Monerau. *Probabilistic Abstract Interpretation*, strony 169–193. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[34] P. Curien, G. Ghelli. On confluence for weakly normalizing systems. *RTA-91*, strony 215–225, 1991.

[35] S. K. Debray, P. L. García, M. Hermenegildo, N.-W. Lin. Estimating the computational cost of logic programs. *Static Analysis*, strony 255–265. Springer, 1994.

[36] D. E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, 1976.

[37] S. Destercke, D. Dubois. The role of generalised p-boxes in imprecise probability models. T. Augustin, F. P. A. Coolen, S. Moral, M. C. M. Troffaes, redaktorzy, *ISIPTA'09: Proceedings of the Sixth International Symposium on Imprecise Probability: Theories and Applications*, strony 179–188, Durham, UK, Lip. 2009. SIPTA.

[38] J. Dhaene, M. Denuit, M. J. Goovaerts, R. Kaas, D. Vyncke. The Concept of Comonotonicity in Actuarial Science and Finance: Theory. *Insurance, mathematics & economics*, 31(2):133–161, 2002.

[39] A. Di Pierro, C. Hankin, H. Wiklicky. Probabilistic ÃŐÂż-calculus and quantitative program analysis. wolumen 15, strony 159–179, April 2005.

[40] A. Di Pierro, C. Hankin, H. Wiklicky. *Abstract Interpretation for Worst and Average Case Analysis*, wolumen 4444 serii *LNCS*, strony 160–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[41] A. Di Pierro, P. Sotin, H. Wiklicky. Relational Analysis and Precision via Probabilistic Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 220(3):23–42, 2008.

[42] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, 1976.

[43] G. J. Duck, P. J. Stuckey, M. J. G. de la Banda, C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. B. Demoen, V. Lifschitz,

redaktorzy, *Proc. Logic Programming, 20th International Conference, ICLP 2004*, wolumen 3132 serii *LNCS*, strony 90–104. Springer, 2004.

[44] G. J. Duck, P. J. Stuckey, M. Sulzmann. Observable Confluence for Constraint Handling Rules. V. Dahl, I. Niemelä, redaktorzy, *ICLP*, wolumen 4670 serii *LNCS*, strony 224–239. Springer, 2007.

[45] R. Durbin, S. Eddy, A. Krogh, G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999.

[46] S. Ferson. Model uncertainty in risk analysis. Tech. report, Centre de Recherches de Royallieu, Universite de Technologie de Compiegne, 2014.

[47] S. Ferson, J. Hajagos, D. Berleant, J. Zhang, W. T. Tucker, L. Ginzburg, W. Oberkampf. Dependence in Dempster-Shafer theory and probability bounds analysis. *New York*, 2004.

[48] S. Ferson, V. Kreinovich, L. Ginzburg, D. S. Myers, K. Sentz. Constructing Probability Boxes and Dempster-Shafer Structures. Sand2002-4015, Sandia National Laboratories, 2002.

[49] S. Ferson, V. Kreinovich, L. Ginzburg, D. S. Myers, K. Sentz. Constructing Probability Boxes and Dempster-Shafer Structures. *Small*, (January):143, 2003.

[50] L. M. F. Fioriti, H. Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. *POPL 2015*, strony 489–501, 2015.

[51] P. Flajolet, B. Salvy, P. Zimmermann. Automatic Average-Case Analysis of Algorithm. *Theor. Comput. Sci.*, 79(1):37–109, 1991.

[52] V. Forejt, M. Z. Kwiatkowska, G. Norman, D. Parker. Automated Verification Techniques for Probabilistic Systems. M. Bernardo, V. Issarny, redaktorzy, *SFM*, wolumen 6659 serii *LNCS*, strony 53–113. Springer, 2011.

[53] W. D. Frazer, A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, 1970.

[54] T. W. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[55] T. W. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, Sier. 2009.

[56] T. W. Frühwirth, A. D. Pierro, H. Wiklicky. Probabilistic Constraint Handling Rules. *Electr. Notes Theor. Comput. Sci.*, 76:115–130, 2002.

[57] A. Gao. *Modular average case analysis: Language implementation and extension*. Ph.d. thesis, University College Cork, 2013.

[58] J. Geldenhuys, M. B. Dwyer, W. Visser. Probabilistic symbolic execution. *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, strony 166–176. ACM, 2012.

[59] Gerald B. Folland. *Guide to Advanced Real Analysis*. Mathematical Association of America, Washington, 2009.

[60] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, R. Thiemann. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, Jan 2017.

[61] J. Gordon, E. H. Shortliffe. The Dempster-Shafer Theory of Evidence. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, strona 21 pp, 1984.

[62] X. Guo, M. Boubekeur, J. McEnery, D. Hickey. ACET based scheduling of soft real-time systems: An approach to optimise resource budgeting. *International Journal of Computers and Communications*, 1(1):82–86, 2007.

[63] R. Haemmerlé. Diagrammatic confluence for Constraint Handling Rules. *TPLP*, 12(4-5):737–753, 2012.

[64] N. Halbwachs. Delay analysis in synchronous programs. *Fifth Conference on Computer-Aided Verification*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[65] N. Halbwachs, Y. Proy, P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[66] S. Hart, M. Sharir, A. Pnueli. Termination of probabilistic concurrent program. *ACM Transactions on Programming ...*, 5(3):356–380, 1983.

[67] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, jun 1990.

[68] J. R. Hindley. An abstract Church-Rosser theorem. II: applications. *J. Symb. Log.*, 39(1):1–21, 1974.

[69] T. S. Y. Ho. *The Oxford guide to financial modeling, applications for capital markets, corporate finance, risk management and financial institutions*. Oxford University Press, New York, 2004.

[70] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 27(4):797–821, 1980.

[71] A. Itai. A randomized algorithm for checking equivalence of circular lists. *Inf. Process. Lett.*, 9(3):118–121, 1979.

[72] J. Jacod, P. Protter. *Probability Essentials*. Universitext. Springer-Verlag Berlin Heidelberg, wydanie 2, 2004.

[73] T. Jech. *Set theory*. 2006.

[74] B. Kafle, J. P. Gallagher. Convex polyhedral abstractions, specialisation and property-based predicate splitting in Horn clause verification. *Electronic Proceedings in Theoretical Computer Science*, 169(318337):53–67, 2014.

[75] D. R. Karger, C. Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996.

[76] S. Kerrison, K. Eder. Energy modelling and optimisation of software for a hardware multi-threaded embedded microprocessor. *University of Bristol, Bristol, Tech. Rep*, 2013.

[77] M. H. Kirkeby, H. Christiansen. Confluence and convergence in probabilistically terminating reduction systems. *Logic-Based Program Synthesis and Transformation - 27th International Symposium, LOPSTR 2017, Namur, Belgium, October 10-12, 2017*, wolumen abs/1709.05123, 2017. (accepted for publication).

[78] M. H. Kirkeby, M. Rosendahl. Probabilistic resource analysis by program transformation. M. van Eekelen, U. Dal Lago, redaktorzy, *Foundational and*

*Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, strony 60–80, Cham, 2016. Springer International Publishing.

[79] S. Kirkpatrick, D. G. Jr., M. P. Vecchi. Optimization by simmulated annealing. *Science*, 220(4598):671–680, 1983.

[80] J. Knoop, L. Kovács, J. Zwirchmayr. *Symbolic Loop Bound Computation for WCET Analysis*, strony 227–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[81] A. Kolmogoroff. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1933.

[82] D. Kozen. Semantics of probabilistic programs. *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, 350:328–350, 1979.

[83] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328 – 350, 1981.

[84] M. Kwiatkowska, G. Norman, D. Parker. Advances and challenges of probabilistic model checking. *48th Annual Allerton Conference on Communication, Control, and Computing*, strony 1691–1698. IEEE.

[85] J. Langbein, F. Raiser, T. W. Frühwirth. A State Equivalence and Confluence Checker for CHRs. P. V. Weert, L. D. Koninck, redaktorzy, *Proceedings of the 7th International Workshop on Constraint Handling Rules*, Report CW 588, strony 1–8. Katholieke Universiteit Leuven, Belgium, 2010.

[86] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, K. Eder. *Energy Consumption Analysis of Programs Based on XMOS ISA-Level Models*, strony 72–90. Springer International Publishing, Cham, 2014.

[87] P. López-Garcia, L. Darmawan, F. Bueno. A Framework for Verification and Debugging of Resource Usage Properties: Resource Usage Verification. M. V. Hermenegildo, T. Schaub, redaktorzy, *ICLP (Technical Communications)*, wolumen 7 serii *LIPIcs*, strony 104–113. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[88] P. López-García, L. Darmawan, M. Klemen, U. Liqat. Interval-based resource usage verification by translation into horn clauses and an application to energy consumption. *TPLP*, 18(2):167–223, 2018.

[89] E. Maffioli, M. G. Speranza, C. Vercellis. Randomized algorithms: An annotated bibliography. *Annals of Operations Research*, 1(3):331–345, 1984.

[90] Merriam-Webster. Definition of Probability.

[91] D. L. Métayer. ACE: an automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.

[92] A. Miné. A new numerical abstract domain based on difference-bound matrices. O. Danvy, A. Filinski, redaktorzy, *Programs as Data Objects*, strony 155–172, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[93] A. Miné. A few graph-based relational numerical abstract domains. M. V. Hermenegildo, G. Puebla, redaktorzy, *Static Analysis, 9th International Symposium, SAS 2002, Madrid, Spain, September 17-20, 2002, Proceedings*,

wolumen 2477 serii *Lecture Notes in Computer Science*, strony 117–132. Springer, 2002.

[94] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[95] D. Monniaux. Abstract Interpretation of Probabilistic Semantics. J. Palsberg, redaktor, *SAS*, wolumen 1824 serii *LNCS*, strony 322–339. Springer, 2000.

[96] D. Monniaux. Backwards Abstract Interpretation of Probabilistic Programs. *European Symposium on Programming*, strony 367–382, 2001.

[97] D. Monniaux. Abstract interpretation of programs as Markov decision processes. *Science of Computer Programming*, 58(1-2):179–205, oct 2005.

[98] I. Montes, S. Destercke. Comonotonicity for sets of probabilities. *Fuzzy Sets and Systems*, 328:1–34, dec 2017.

[99] I. Montes, E. Miranda, R. Pelessoni, P. Vicig. Sklar's theorem in an imprecise setting. *Fuzzy Sets and Systems*, strony 1–23, 2014.

[100] R. E. Moore. *Interval analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1966.

[101] C. Morgan, A. McIver, K. Seidel. Probabilistic Predicate Transformers. *ACM Trans. Program. Lang. Syst.*, 18(3):325–353, 1996.

[102] R. Motwani, P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[103] J. A. Navas, E. Mera, P. López-García, M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. V. Dahl, I. Niemelä, redaktorzy, *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, wolumen 4670 serii *Lecture Notes in Computer Science*, strony 348–363. Springer, 2007.

[104] R. B. Nelsen. Chapter 2 - Definitions and Basic Properties. *An Introduction to Copulas*, strony 1–43, 2007.

[105] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[106] H. T. Nguyen. On random sets and belief functions. *Journal of Mathematical Analysis and Applications*, 65(3):531–542, oct 1978.

[107] M. Oulamara, A. J. Venet. Abstract interpretation with higher-dimensional ellipsoids and conic extrapolation. D. Kroening, C. S. Pasareanu, redaktorzy, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, wolumen 9206 serii *Lecture Notes in Computer Science*, strony 415–430. Springer, 2015.

[108] A. D. Pierro, H. Wiklicky. Probabilistic data flow analysis: a linear equational approach. *Proceedings Fourth International Symposium*, wolumen 119, strony 150–165. Open Publishing Association, 2013.

[109] H. S. Pollman, M. Carro, P. López-Garcia. Probabilistic Cost Analysis of Logic Programs: A First Case Study. *INGENIARE - Revista Chilena de Ingeniera*, 17(2):195–204, 2009.

[110] M. O. Rabin. The choice coordination problem. *Acta Informatica*, 17(2):121–134, 1982.

[111] F. Raiser, H. Betz, T. W. Frühwirth. Equivalence of CHR States Revisited. F. Raiser, J. Sneyers, redaktorzy, *Proc. 6th International Workshop on Constraint Handling Rules*, Report CW 555, strony 33–48. Katholieke Universiteit Leuven, Belgium, 2009.

[112] F. Raiser, P. Tacchella. On Confluence of Non-terminating CHR Programs. K. Djelloul, G. J. Duck, M. Sulzmann, redaktorzy, *Constraint Handling Rules, 4th Workshop, CHR 2007*, strony 63–76, Porto, Portugal, 2007.

[113] U. K. Rakowsky. Fundamentals of the Dempster-Shafer Theory and its applications to reliability modeling. *International Journal of Reliability, Quality and Safety Engineering*, 14(06):579–601, dec 2007.

[114] H. Riis Nielson, F. Nielson. *Semantics with Applications: an Appetizer*. 2007.

[115] M. Rosendahl. *Automatic program analysis (Master's thesis)*. Praca doktorska, University of Copenhagen, 1986.

[116] M. Rosendahl. Automatic complexity analysis. *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, strony 144–156, New York, New York, USA, 1989. ACM Press.

[117] M. Rosendahl. Simple Driving Techniques. T. Æ. Mogensen, D. A. Schmidt, I. H. Sudborough, redaktorzy, *The Essence of Computation*, wolumen 2566 serii *LNCS*, strony 404–419. Springer, 2002.

[118] M. Rosendahl, M. H. Kirkeby. Probabilistic Output Analysis by Program Manipulation. *Electronic Proceedings in Theoretical Computer Science*, 194(318337):110–124, 2015.

[119] W. Rudin. *Real and complex analysis*. 2006.

[120] S. Sankaranarayanan, A. Chakarov, S. Gulwani. Static analysis for probabilistic programs. *ACM SIGPLAN Notices*, 48(6):447, jun 2013.

[121] T. Sato. A statistical learning method for logic programs with distribution semantics. *ICLP 1995*, strony 715–729, 1995.

[122] T. Sato. A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems*, 31(2):161–176, 2008.

[123] T. Sato, P. J. Meyer. Infinite probability computation by cyclic explanation graphs. *TPLP*, 14(6):909–937, 2014.

[124] M. Schellekens. *A modular calculus for the average cost of data structuring, efficiency-oriented programming in MOQA*. Springer, New York London, 2008.

[125] L. Schor, I. Bacivarov, H. Yang, L. Thiele. Worst-Case Temperature Guarantees for Real-Time Applications on Multi-core Systems. M. D. Natale, redaktor, *IEEE Real-Time and Embedded Technology and Applications Symposium*, strony 87–96. IEEE, 2012.

[126] T. Schrijvers, T. W. Frühwirth. Analysing the CHR Implementation of Union-Find. A. Wolf, T. W. Frühwirth, M. Meister, redaktorzy, *W(C)LP*, wolumen 2005-01 serii *Ulmer Informatik-Berichte*, strony 135–146. Universität Ulm, Germany, 2005.

[127] A. Serrano, P. López-García, M. V. Hermenegildo. Resource usage analysis of logic programs via abstract interpretation using sized types. *TPLP*, 14(4-5):739–754, 2014.

[128] R. Sethi. Testing for the Church-Rosser Property. *J. ACM*, 21(4):671–679, 1974.

[129] J. Sneyers, W. Meert, J. Vennekens, Y. Kameya, T. Sato. CHR(PRISM)-based Probabilistic Logic Learning. *TPLP*, 10(4–6), 2010.

[130] J. Sneyers, D. D. Schreye. Probabilistic termination of CHRiSM programs. *LOPSTR 2011*, strony 221–236, 2011.

[131] J. Sneyers, P. V. Weert, T. Schrijvers, L. D. Koninck. As time goes by: Constraint Handling Rules. *TPLP*, 10(1):1–47, 2010.

[132] A. Takahito, N. Hirokawa, J. Nagele. Confluence Competition, 2018.

[133] R. E. Tarjan, J. van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, mar 1984.

[134] S. J. Taylor. *Introduction to Measure and Integration*. Cambridge University Press, 1973.

[135] V. Tiwari, S. Malik, A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[136] A. Uwimbabazi. *Extended Probabilistic Symbolic Execution*. Praca doktorska, Stellenbosch University, 2013.

[137] V. van Oostrom. Confluence by Decreasing Diagrams. *Theor. Comput. Sci.*, 126(2):259–280, 1994.

[138] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, apr 1967.

[139] D. Volpano, C. Irvine, G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167, 1996.

[140] P. Walley. Measures of uncertainty in expert systems. *Artificial Intelligence*, 83(1):1–58, 1996.

[141] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[142] M. Weiser. Program slicing. *Software Engineering, IEEE Transactions on*, SE-10(4):352–357, July 1984.

[143] E. W. Weisstein. *q-Pochhammer Symbol*. MathWorld – A Wolfram Web Resource, 2017.

[144] A. Wierman, L. L. H. Andrew, A. Tang. Stochastic Analysis of Power-Aware Scheduling. *Proceedings of Allerton Conference on Communication, Control and Computing*. Urbana-Champaign, IL, 2008.

[145] N. Wilson. Algorithms for Dempster-Shafer Theory. *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, strony 421–475. Springer Netherlands, Dordrecht, 2000.

[146] S. Wolfram. *The Mathematica, book*. Wolfram Media Cambridge University Press, Cambridge, wydanie 3. ed., 1996.

[147] R. R. Yager, L. Liu. *Classic Works of the Dempster-Shafer Theory of Belief Functions*. 2008.

[148] R. Zippel. Probabilistic algorithms for sparse polynomials. *EUROSAM 1979*, strony 216–226, 1979.

[149] F. Zuleger, S. Gulwani, M. Sinn, H. Veith. *Bound Analysis of Imperative Programs with the Size-Change Abstraction*, strony 280–297. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

RECENT RESEARCH REPORTS

#150 Xueliang Li. *Detecting, Diagnosing and Fixing Energy Issues for Mobile Applications*. PhD thesis, Roskilde, Denmark, June 2017.

#149 Benedicte Fleron. *Participation in the Implementation of IT Support for Work Practices at 4 Emergency Departments: Ethnographic Studies of an Improvisational IT Implementation Project*. PhD thesis, Roskilde, Denmark, October 2016.

#148 Bishoksan Kafle. *Components for Automatic Horn Clause Verification*. PhD thesis, Roskilde, Denmark, October 2016.

#147 Maria Ie Manikas. *Pilot Implementations: Enacted, Embedded, Relational, Multiple*. PhD thesis, Roskilde, Denmark, June 2016.

#146 Jesper B. Berger. *E-Government Harm: An Assessment of the Danish Coercive Digital Post Strategy*. PhD thesis, Roskilde, Denmark, June 2015.

#145 John P. Gallagher, Mai Ajspur, and Bishoksan Kafle. An optimised algorithm for determinisation and completion of finite tree automata. 25 pp. September 2014, Roskilde University, Roskilde, Denmark.

#144 Magnus Rotvit Perlt Hansen. *Discovering the Process of User Expectating in a Pilot Implementation Expectations and Experiences in Information Systems Development*. PhD thesis, Roskilde, Denmark, June 2014.

#143 Keld Helsgaun. Solving the Bottleneck Traveling Salesman Problem Using the Lin-Kernighan-Helsgaun Algorithm. 42 pp. May 2014, Roskilde University, Roskilde, Denmark.

#142 Keld Helsgaun. Solving the Clustered Traveling Salesman Problem Using the Lin-Kernighan-Helsgaun Algorithm. 13 pp. May 2014, Roskilde University, Roskilde, Denmark.

#141 Keld Helsgaun. Solving the Equality Generalized Traveling Salesman Problem Using the Lin-Kernighan-Helsgaun Algorithm. 15 pp. May 2014, Roskilde University, Roskilde, Denmark.

#140 Anders Barlach. *Effekt-drevet IT udvikling Eksperimenter med effektdrevne systemudviklingsprojekter, der involverer CSC Scandihealth og kunder fra det danske sundhedsvæsen*. PhD thesis, Roskilde, Denmark, November 2013.

#139 Mai Lise Ajspur. *Tableau-based Decision Procedures for Epistemic and Temporal Epistemic Logics*. PhD thesis, Roskilde, Denmark, October 2013.

#138 Rasmus Rasmussen. *Electronic Whiteboards in Emergency Medicine Studies of Implementation Processes and User Interface Design Evaluations*. PhD thesis, Roskilde, Denmark, April 2013.